FORTRAN-80

APPLE II
or
APPLE II PLUS

MICROSOFT

MICROSOFT
CONSUMER PRODUCTS

Non Disclosure Agreement and Registration
FORTRAN-80
for APPLE II

IMPORTANT: PLEASE COMPLETE THIS FORM IMMEDIATELY. In order to provide you any updates or fixes, we must have your completed form on file. Failure to register and sign the non-disclosure agreement voids any warranty expressed or implied.

Purchased from:

Company

Address

City & State Zip

Phone

Purchased by:

Name

Company

Address

City & State Zip

Customer's hardware description (brands, configuration, amount of memory, etc.):

Operating System: Version:

NOTE: THIS NON-DISCLOSURE STATEMENT MUST BE SIGNED TO COMPLETE THIS REGISTRATION FORM. The party below agrees that it is receiving a copy of Microsoft FORTRAN-80 for use on a single computer only, as designated above. The party agrees that all copies of FORTRAN-80 are owned by Microsoft, that all copies will display Microsoft's copyright, that all copies will be strictly safeguarded against disclosure to or use by persons not authorized by Microsoft to use FORTRAN-80, and that the location of all copies will be reported to Microsoft at Microsoft's request. The party may make up to two additional copies only, for back-up purposes. The party agrees that unauthorized copying or disclosure will cause great damage to Microsoft.

Signed Date

Title

400 108th Ave. N.E., Suite 200 • Bellevue, WA 98004 • (206) 454-1315

Addendum to: FORTRAN-80 User's Guide for the Apple II Computer

Add to Section 1.4.2, Create the Source Program

After you have finished entering the DEMO program, enter a CTRL <Z>. When the asterisk (*) appears on the screen, enter an <E> to exit the text editor and save the source file.

FORTRAN-80 User's Manual

CONTENTS

Chapter 1 Overview

1.1 Introduction
1.2 The FORTRAN-80 System Diskette
1.3 Making a Backup Diskette
1.3.1 Backup Procedure for Single-Drive System
1.3.2 Backup Procedure for Multi-Drive System
1.4 FORTRAN Program Development Steps
1.4.1 Sample Session for Single-Drive System
1.4.2 Sample Session for Multi-Drive System

Chapter 2 FORTRAN-80 Compiler

2.1 Running the Compiler
2.2 Command Format
2.2.1 FORTRAN-80 Command Switches
2.3 FORTRAN-80 Compiler Error Messages

Chapter 3 LINK-80 Linking Loader

3.1 Running LINK-80
3.2 Using LINK-80
3.3 Additional LINK-80 Features
3.3.1 The Reset Switch (/R)
3.3.2 The Search Switch (/S)
3.3.3 Special Switches
3.4 LINK-80 Error Messages

Chapter 4 Executing FORTRAN Programs

4.1 FORTRAN-80 Runtime Error Messages

Chapter 5 FORTRAN-80 I/O

5.1 Disk Files
5.1.1 Random Disk I/O

Appendix A Transferring Files with a Single-Drive System

Appendix B Programs in ROM

CHAPTER 1

OVERVIEW

1.1 INTRODUCTION

The purpose of this FORTRAN-80 User's Manual is to give you practical information about getting a FORTRAN-80 program up and running on your Apple II computer. All the steps necessary to use FORTRAN-80 -- compiling, loading, executing, etc. -- are described in the following pages.

Some Apple II computer users have a single-drive system and others have a multi-drive system. When necessary, this manual outlines a separate procedure for each type of system.

This manual assumes that you know how to write a FORTRAN program or are in the process of learning FORTRAN. If you need extra help with the FORTRAN language, we recommend the following texts:

1. "FORTRAN IV, A Self-Teaching Guide" by Friedmann, Greenberg, and Hoffberg (Wiley, 1975)

2. "Ten Statement FORTRAN Plus FORTRAN IV" by Kennedy and Solomon (Prentice-Hall, 1975, Second Edition)

3. "Guide to FORTRAN-IV Programming" by Daniel McCracken (Wiley, 1965)

1.2 THE FORTRAN-80 SYSTEM DISKETTE

The FORTRAN-80 system from Microsoft is complete on one diskette. Microsoft provides you with a 13-sector diskette and a 16-sector diskette for your convenience. Use the diskette that is appropriate for your system. The files on the FORTRAN-80 system diskette are:

1. F80.COM -- FORTRAN-80 Compiler

2. L80.COM -- LINK-80 Linking Loader

3. FORLIB.REL -- FORTRAN-80 Runtime Library

4. PI.FOR -- A FORTRAN demonstration program

5. RANTST.FOR -- A FORTRAN test program

6. Source Files, FORTRAN-80 Runtime I/O:

   - DSKDRV.MAC
   - TTYDRV.MAC
   - LPTDRV.MAC
   - INIT.MAC
   - IOINIT.MAC
   - LUNTB.MAC

The FORTRAN-80 compiler is the program that reads your FORTRAN program (source file) and produces a relocatable, binary file (object file) that is machine readable.

The LINK-80 Linking Loader is used to convert the compiled relocatable object version of your program into an absolute version that is properly loaded and ready to be executed. LINK-80 automatically performs the tasks of searching the library and loading the proper subroutines for a given program.

The FORTRAN Library is a set of subroutines that may be called by the compiled program.

PI.FOR and RANTST.FOR are FORTRAN programs that can be compiled and run to check out your system. PI.FOR calculates pi, and RANTST.FOR performs a thorough test of the system.

The FORTRAN-80 runtime I/O routines supplied on the diskette are source files of some of the modules in the FORTRAN library. IF YOU HAVE MICROSOFT'S MACRO-80 ASSEMBLER AND LIB-80 LIBRARY MANAGER, you can modify these I/O routines. MACRO-80 and LIB-80 are not supplied with FORTRAN-80 for the Apple II. They may be obtained by purchasing Microsoft Consumer Products' Assembly Language Development System for the Apple II. Modification of these routines should not be necessary unless you are using unusual I/O devices or placing programs in ROM.

1.3 MAKING A BACKUP DISKETTE

Always work with a backup copy of the FORTRAN-80 system master and store the original for safekeeping. Follow the procedure below to make a backup copy of the master, and then store the original in a safe, dry, non-magnetic place.

The backup procedure is outlined first for a single-drive system (Section 1.3.1) and then for a multi-drive system (Section 1.3.2).

1.3.1 Backup Procedure For Single-Drive System

NOTE: Your distribution diskette does not contain a copy of CP/M, and thus WILL NOT BOOT.

To perform the backup procedure, you will need three diskettes:

-- CP/M system diskette with FORMAT and COPY

-- the FORTRAN-80 master diskette (13- or 16-sector, depending on your system)

-- a fresh, blank diskette

1. First, you must format the blank diskette. To do this, put the CP/M system diskette in the drive and boot it as usual. Enter the command

   ```
   FORMAT A:
   ```

   When you are prompted "Hit RETURN to begin", insert the blank diskette and hit RETURN. When the FORMAT process is complete, place your CP/M system disk back in the drive and hit RETURN to return to CP/M command level.

2. Now you are ready to copy the FORTRAN system onto the blank diskette. To do this, enter

   ```
   COPY A:=A:
   ```

   to begin the CP/M COPY program. When you are asked to insert the MASTER disk, place the FORTRAN-80 master in the drive and hit RETURN.

3. In a few seconds, the disk will stop and you will be asked to insert the SLAVE disk. Remove the FORTRAN-80 master and insert the newly formatted blank diskette, and hit RETURN to continue.

4. After you have repeated the process above three or four times, the computer will announce "COPY complete". Place your CP/M system disk in the drive and hit RETURN to return to CP/M command level.

5. Now that you have made a backup copy of your FORTRAN-80 master disk, you must write a copy of the CP/M system itself onto the backup so that it will boot. With your CP/M system disk in the drive, enter

```
COPY A:=A:/S
```

When you are asked to insert the MASTER disk, leave your CP/M system disk in the drive and hit RETURN. After the disk has stopped, insert your FORTRAN-80 backup disk and hit RETURN. When finished, just hit RETURN to exit to CP/M command level.

1.3.2 Backup Procedure For Multi-Drive System

NOTE: Your distribution diskette does not contain a copy of CP/M, and thus WILL NOT BOOT.

To perform the backup procedure, you will need three diskettes:

-- CP/M system diskette with FORMAT and COPY

-- the FORTRAN-80 master diskette (13- or 16-sector, depending on your system)

-- a fresh, blank diskette

1. First you must format the blank diskette. To do this, put the CP/M system diskette in drive A: and boot it as usual. Place the blank diskette in drive B: and type

   FORMAT B:

   Hit RETURN to begin the FORMAT process. When the FORMAT process is complete, hit RETURN to return to CP/M command level.

2. Now you are ready to copy the FORTRAN system onto the blank diskette. To do this, enter

   COPY B:=A:

   to begin the CP/M COPY program. Remove your CP/M system disk and place the FORTRAN-80 master disk in drive A:. Make sure your newly formatted diskette is in drive B:, and hit RETURN to begin the COPY process.

3. When the computer announces "COPY complete", remove the FORTRAN-80 master from drive A: and insert your CP/M system disk. Then hit RETURN to exit to CP/M.

4. Now that you have made a backup copy of your FORTRAN-80 master disk, you must write a copy of the CP/M system itself onto the backup so that it will boot. Make sure you have your CP/M system disk in drive A: and your FORTRAN-80 backup in drive B:, and enter

   COPY B:=A:/S

   When finished, just hit RETURN to exit to CP/M command level.

1.4 FORTRAN PROGRAM DEVELOPMENT STEPS

Your work with FORTRAN-80 will ultimately produce a useful library of FORTRAN programs that perform specific tasks for your application. That is your goal, but how do you accomplish it?

Program development is an ongoing process of many steps. You start by writing, entering and editing the FORTRAN program. Then the program must be compiled, linked and executed. If you get compile-time or run-time errors, you must once again edit the program; then compile, link, and execute the program until it runs successfully.

Your application and the complexity of your FORTRAN programs will, to a large extent, determine your program development habits. However, to give the beginner an overall view of the FORTRAN-80 system, the following sample session is provided. It outlines the edit/compile/link/execute process in step-by-step form.

The sample session is given first for a single-drive system (Section 1.4.1), and then for a multi-drive system (Section 1.4.2). The same FORTRAN program, as shown in Figure 1, is used in both sessions.

1.4.1 Sample Session For Single-Drive System

1. Organize Your Diskettes

Before beginning your FORTRAN program development, organize your diskettes to optimize disk space. Usually FORTRAN program development will require two diskettes. One contains the FORTRAN compiler and associated utility software. The other contains your text editor and program source and object files. Refer to Appendix A for a file-transfer procedure on a single-drive system.

Work Disk	Compiler Disk
.FOR files (source)	F80.COM
.REL files (relocatable object)	L80.COM
.COM files (executable object)	FORLIB.REL
ED.COM	DDT.COM
	FORLIB.REL

2. Create the Source Program

The first step in FORTRAN program development is always to create the source file. The program is typed in and edited using any CP/M text editor. The example below assumes you are using the CP/M editor, ED.

Boot your work diskette in the usual way and enter

```
ED DEMO.FOR
```

to run the editor. When the asterisk prompt appears, enter the command

```
I
```

to begin automatic line number insertion. Now enter the DEMO program as it appears in Figure 1.

Editing Hint for FORTRAN-80 Programs: Except for line number references, carriage control characters, and "C" (for comment), each FORTRAN program line begins in column 7. Rather than hitting the space bar to get to column 7, you can type control-I (tab). This tabs to column 8, which is acceptable to the compiler, since spaces are insignificant.

When you are finished entering the DEMO program, enter

E

to exit the text editor and save the source file. Now there is a file called DEMO.FOR on your work diskette.

3. Check Program Syntax with Trial Compilation

Before you go on, you can check your program for syntax errors with a "quick" compilation. Remove your work diskette and place the FORTRAN compiler disk in the drive, then type Control-C.

ALWAYS REMEMBER TO TYPE CONTROL-C WHEN YOU CHANGE DISKS AT CP/M COMMAND LEVEL.

To run the FORTRAN-80 compiler, type

F80

When the asterisk appears, place your work diskette in the drive.

Do NOT type Control-C after changing disks this time - if you do, you will return to CP/M. There is no need to type Control-C after changing disks once you see the asterisk prompt.

With your work diskette in the drive, enter

,=DEMO

to compile the file DEMO.FOR. This command causes the compiler to suppress output of the listing and object files, thus speeding up the compilation.

If you have compile-time errors in your program, the offending line along with a brief description of the error will be displayed. (See Section 2.3 for a complete list of error messages.) If you get errors during the trial compilation, go back to Step 1 and correct the source file.

4. Compile the Source Program

The next step is to compile the program, this time creating an object file. Place the FORTRAN compiler diskette in the drive, type Control-C, and type F80 to run the compiler. When the prompt appears, put your work diskette in the drive (don't type Control-C) and enter

```
=DEMO
```

to compile the file DEMO.FOR and create the relocatable object file DEMO.REL. Notice that the only difference between this command and the previous command used for the syntax check is that there is no comma before the equal sign.

5. Link the Program

The next program development step is to link the program with the FORTRAN-80 runtime library and create the machine-executable object file. First put the compiler disk in the drive, type Control-C, and enter

```
L80
```

to run the linking loader. When the asterisk appears, insert your work disk and type

```
DEMO,DEMO/N
```

This command loads your program and specifies the name for the executable object file. When finished, LINK-80 will display some loader information and return the asterisk prompt. Don't worry about the list of "undefined globals." These will be defined in the library search, which is performed next.

Put the compiler disk back in the drive and enter

```
FORLIB/S
```

This command searches the FORTRAN library and links your program with the required subroutines. In a minute or so, the disk will stop and more loader information will be displayed. When the prompt appears, insert your work disk and type

```
/E
```

This will save the machine-executable file DEMO.COM on the disk, and exit to CP/M.

6. Run the Program

At CP/M command level, type

```
DEMO
```

to execute the DEMO program. Even though you've been very careful to remove all compile-time errors, you may still get runtime errors when the program is run. Runtime errors are listed in this manual in Section 4.1. If you get runtime errors, it will be necessary to return to Step 1 and edit the program to correct the errors.

This is the end of the sample session for single-drive systems. Skip the next section, which is for multi-drive systems, and proceed to Chapter 2 for specific information on the FORTRAN-80 compiler.

1.4.2 Sample Session For Multi-Drive System

1. Organize Your Diskettes

Before beginning your FORTRAN program development, organize your diskettes to optimize disk space. Usually FORTRAN program development will require two diskettes. One contains the FORTRAN compiler, associated utilities, and your text editor. The other contains your program source and object files.

Work Disk (Drive A:)	Compiler Disk (Drive B:)
.FOR files (source)	F80.COM
.REL files (relocatable object)	L80.COM
.COM files (executable object)	FORLIB.REL
	ED.COM
	DDT.COM
	PIP.COM

2. Create the Source Program

The first step in FORTRAN program development is always to create the source file. The program is typed in and edited using any CP/M text editor. The example below assumes you are using the CP/M editor, ED.

Insert the compiler disk in drive B:. Boot your work disk as usual and type

```
B:ED DEMO.FOR
```

to run the editor. When the asterisk prompt appears, enter the command

```
I
```

to begin automatic line number insertion. Now enter the DEMO program as it appears in Figure 1.

Editing Hint for FORTRAN-80 Programs: Except for line number references, carriage control characters, and "C" (for comment), each FORTRAN program line begins in column 7. Rather than hitting the space bar to get to column 7, you can type control-I (tab). This tabs to column 8, which is acceptable to the compler, since spaces are insignificant.

When you are finished entering the DEMO program, type

```
E
```

to exit the text editor and save the source file on your work diskette. Now there is a file called DEMO.FOR on your work diskette.

3. Check Program Syntax with Trial Compilation

Before you go on, you can check your program for syntax errors with a "quick" compilation. At CP/M command level, type

```
B:F80 ,=DEMO
```

to compile the file DEMO.FOR on your work disk. This command line causes the compiler to suppress output of the listing and object files, thus speeding up the compilation.

If you have compile-time errors in your program, the offending line along with a brief description of the error will be displayed. (See Section 2.3 for a complete list of error messages.) If you get errors during the trial compilation, go back to Step 1 and correct the source file.

4. Compile the Source Program

The next step is to compile the program, this time creating an object file. At CP/M command level, type

```
B:F80 =DEMO
```

to compile the file DEMO.FOR and create the object file DEMO.REL. Notice that the only difference between this command and the previous command used for the syntax check is that there is no comma before the equal sign.

5. Link the Program

The next program development step is to link the program with the FORTRAN-80 runtime library and create the machine-executable object file. Type

```
B:
```

to log on to drive B:. Then type

```
L80 A:DEMO,A:DEMO/N/E
```

This command does the following: it links the object file with the runtime system, producing the executable object file. This file -- called DEMO.COM -- is saved on the diskette in drive A:.

6. Run the Program

At CP/M command level, log on to drive A: and type

```
DEMO
```

to execute the DEMO program. Even though you've been very careful to remove all compile-time errors, you may still get runtime errors when the program is run. Runtime errors are listed in this manual in Section 4.1. If you get runtime errors, it will be necessary to return to Step 1 and edit the program to correct the errors.

FIGURE 1. DEMO.FOR
FORTRAN SOURCE PROGRAM

```
C
C  *** ROD'S COLOR PATTERN ***
C
C  BASED ON APPLE INTEGER BASIC
C  PROGRAM ON PAGE 55 OF JAN 78
C   APPLE ][ REFERENCE MANUAL
C
      PROGRAM DEMO

      BYTE W,I,J,K,L,M

      CALL GR(1,0)

10    DO 20 W = 3,50
      DO 20 I = 1,19
      DO 20 J = 0,19

      K = I + J
      CALL COLOR(J*3/(I+3)+I*W/12)

      L = 40 - K
      M = 40 - I

      CALL PLOT(I,K)
      CALL PLOT(K,I)
      CALL PLOT(L,M)
      CALL PLOT(M,L)
      CALL PLOT(K,M)
      CALL PLOT(M,K)
      CALL PLOT(I,L)
      CALL PLOT(L,I)

      IF (INKEY(0)) GO TO 30

20    CONTINUE

30    CALL TEXT
      STOP *BYE*

      END
```

CHAPTER 2

FORTRAN-80 COMPILER

If you followed the sample session, you are becoming familiar with the software in the FORTRAN-80 system. Now let's look specifically at the FORTRAN-80 compiler.

2.1 RUNNING THE COMPILER

When you put the FORTRAN system diskette in the drive and type

```
F80
```

at CP/M command level, you are invoking the FORTRAN-80 compiler.

The compiler prompts with an asterisk. To exit the compiler and return to CP/M, type control-C.

You may also type the compiler command on the same line as the invocation. This is called a "command line." Example:

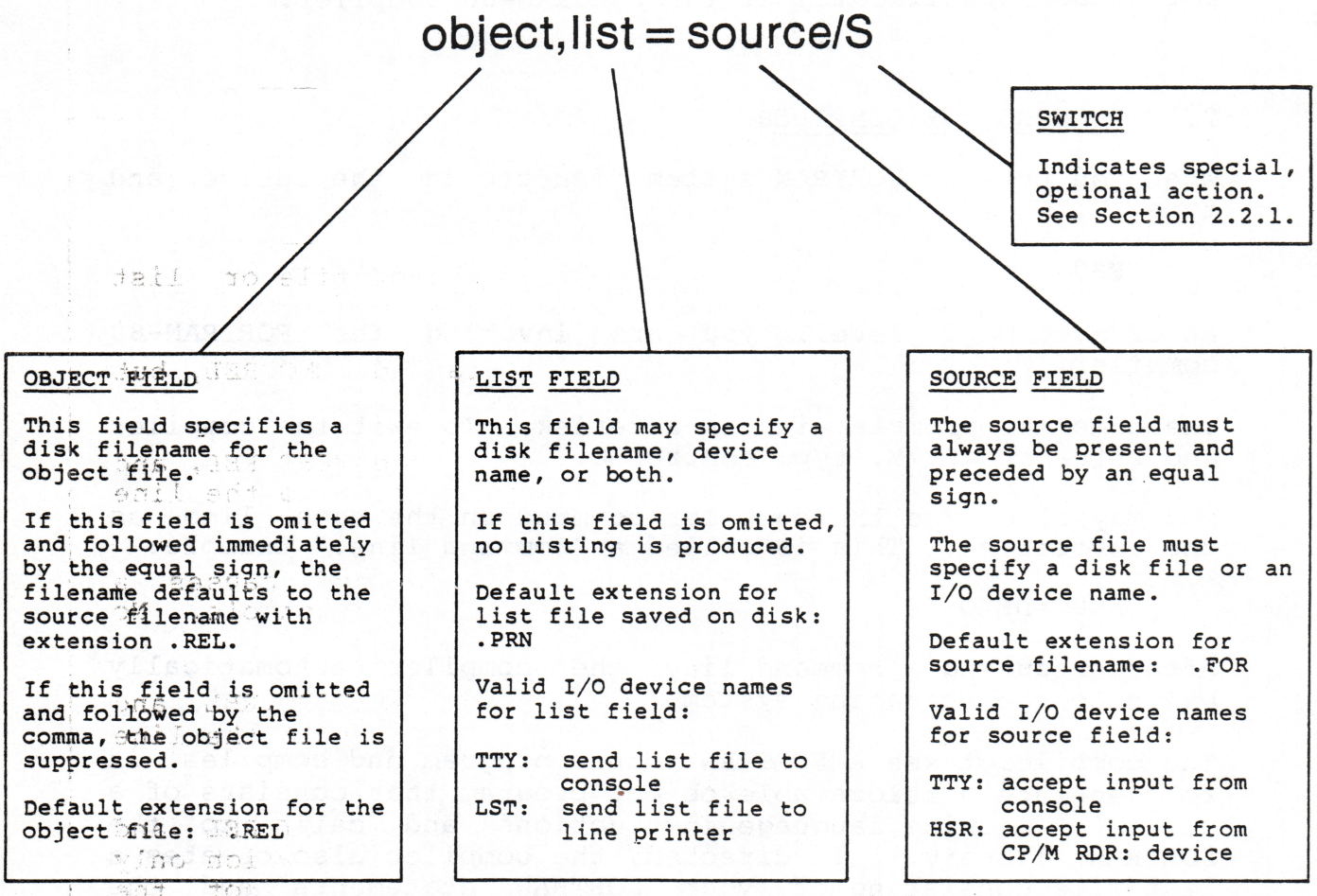
```
F80 =DEMO
```

After executing a command line, the compiler automatically exits to the operating system.

The compiler takes a FORTRAN source program and compiles it to generate a relocatable object program that consists of a series of machine language instructions and calls to the FORTRAN library. If directed, the compiler also creates a list file consisting of your FORTRAN statements and the machine code generated from them. This list file, unlike the REL (relocatable object) program, can be listed and printed from CP/M command level.

2.2 COMMAND FORMAT

The compiler command format is very flexible. The command not only tells the name of the source file to be compiled, but which compile-time options you want to use as well. These options include creating a list file, creating an object file, the names and extensions for these files, what devices they go to, and what device the source file is coming from. That's a lot for one command, but here's how it works:


object,list = source/S
SWITCH
Indicates special, optional action. See Section 2.2.1.
OBJECT FIELD
This field specifies a disk filename for the object file.
If this field is omitted and followed immediately by the equal sign, the filename defaults to the source filename with extension .REL.
If this field is omitted and followed by the comma, the object file is suppressed.
Default extension for the object file: .REL
LIST FIELD
This field may specify a disk filename, device name, or both.
If this field is omitted, no listing is produced.
Default extension for list file saved on disk: .PRN
Valid I/O device names for list field:
TTY: send list file to console
LST: send list file to line printer
SOURCE FIELD
The source field must always be present and preceded by an equal sign.
The source file must specify a disk file or an I/O device name.
Default extension for source filename: .FOR
Valid I/O device names for source field:
TTY: accept input from console
HSR: accept input from CP/M RDR: device

SUMMARY

Default extensions:	.FOR	FORTRAN source file
	.REL	Relocatable object file
	.PRN	List file
Device Names:	LST:	Line printer
	TTY:	Console
	HSR:	High speed reader
Switches:	O	List with octal
	H	List with hex
	N	Suppress object in list
	R	Force object file
	P	Extra stack space
	M	Compiled code for ROM
CP/M filename format:	[A:-F:]filename[.ext]	

Examples of Compiler Commands

(In each example, the source file is DEMO.FOR)

,=DEMO	Syntax check only. No object file or list file is created.
=DEMO	Creates an object file called DEMO.REL, but no list file.
TEST,LST:=DEMO	Creates an object file called TEST.REL, and causes a listing to be sent to the line printer.
,TTY:=B:DEMO	Compiles the file B:DEMO.FOR and causes a listing to be sent to the console. No object file is produced.
,LST:=DEMO/R	Creates an object file called DEMO.REL and causes a listing to be sent to the line printer.
TEST,TEST=DEMO/N	Creates an object file called TEST.REL and a list file called TEST.PRN in which only the source statements are listed (not the generated object code). The list file is sent to the disk.

2.2.1 FORTRAN-80 Command Switches

A switch on the end of a compiler command specifies a special action to be taken during compilation. Switches are always preceded by a slash (/). More than one switch may be used in the same command. The available switches are:

Switch	Action
O	Print all listing addresses in octal.
H	Print all listing addresses in hexadecimal (default condition).
N	Do not list the object code that is generated. List only the FORTRAN source code.
R	Force generation of a relocatable object file. The /R switch overrides a command that suppresses the object file.
P	Each /P allocates an extra 256 bytes of stack space for use during compilation. Use /P if stack overflow errors occur during compilation. Otherwise not needed.
M	Specifies to the compiler that the generated code should be in a form which can be loaded into ROMs. See Appendix B.

2.3 FORTRAN-80 COMPILER ERROR MESSAGES

The FORTRAN-80 Compiler detects two kinds of errors: Warnings and Fatal Errors. When a Warning is issued, compilation continues with the next item on the source line. When a Fatal Error is found, the compiler ignores the rest of the logical line, including any continuation lines. Warning messages are preceded by percent signs (%), and Fatal Errors by question marks (?). The editor line number, if any, or the physical line number is printed next. It is followed by the error code or error message and the last 20 characters scanned at the time the error was detected.

Example:

```
?Line 25: Mismatched Parentheses: I=(I+J))
%Line 16: Missing Integer Variable: I(2*2,
```

When either type of error occurs, the program should be changed so that it compiles without errors. No guarantee is made that a program that compiles with errors will execute sensibly.

Fatal Errors:

100	Illegal Statement Number
101	Statement Unrecognizable or Misspelled
102	Illegal Statement Completion
103	Illegal DO Nesting
104	Illegal Data Constant
105	Missing Name
106	Illegal Procedure Name
107	Invalid DATA Constant or Repeat Factor
108	Incorrect Number of DATA Constants
109	Incorrect Integer Constant
110	Invalid Statement Number
111	Not a Variable Name
112	Illegal Logical Form Operator
113	Data Pool Overflow
114	Literal String Too Large
115	Invalid Data List Element in I/O
116	Unbalanced DO Nest
117	Identifier Too Long
118	Illegal Operator
119	Mismatched Parenthesis
120	Consecutive Operators
121	Improper Subscript Syntax
122	Illegal Integer Quantity
123	Illegal Hollerith Construction
124	Backwards DO reference
125	Illegal Statement Function Name
126	Illegal Character for Syntax
127	Statement Out of Sequence
128	Missing Integer Quantity
129	Invalid Logical Operator
130	Illegal Item Following INTEGER or REAL or LOGICAL
131	Premature End Of File on Input Device
132	Illegal Mixed Mode Operation
133	Function Call with No Parameters
134	Stack Overflow
135	Illegal Statement Following Logical IF
136	Wrong Number of Subscripts
137	File Not Found

Warnings:

0	Duplicate Statement Label
1	Illegal DO Termination
2	Block Name = Procedure Name
3	Array Name Misuse
4	COMMON Name Usage
5	Wrong Number of Subscripts
6	Array Multiply EQUIVALENCEd within a Group
7	Multiple EQUIVALENCE of COMMON
8	COMMON Base Lowered
9	Non-COMMON Variable in BLOCK DATA
10	Empty List for Unformatted WRITE
11	Non-Integer Expression
12	Operand Mode Not Compatible with Operator
13	Mixing of Operand Modes Not Allowed
14	Missing Integer Variable
15	Missing Statement Number on FORMAT
16	Zero Repeat Factor
17	Zero Format Value
18	Format Nest Too Deep
19	Statement Number Not FORMAT Associated
20	Invalid Statement Number Usage
21	No Path to this Statement
22	Missing Do Termination
23	Code Output in BLOCK DATA
24	Undefined Labels Have Occurred
25	RETURN in a Main Program
26	STATUS Error on READ
27	Invalid Operand Usage
28	Function with no Parameter
29	Hex Constant Overflow
30	Division by Zero
32	Array Name Expected
33	Illegal Argument to ENCODE/DECODE

CHAPTER 3

LINK-80 LINKING LOADER

As demonstrated in the Sample Session, compiled FORTRAN programs are loaded into memory, linked, and executed using the LINK-80 linking loader. The linking loader converts the compiled relocatable object version of your program into an absolute version that is executable. It automatically combines the required portions of the FORTRAN runtime library with your object program. The file containing the executable memory image (command file) produced by LINK-80 can be saved on disk and run at a later time.

3.1 RUNNING LINK-80

To run LINK-80, type L80. LINK-80 prompts with an asterisk to indicate command level. To exit to CP/M from L80 command level, type /E (or control-C).

3.2 USING LINK-80

The operations you will perform most often with LINK-80 are loading a relocatable object (.REL) file and saving an executable (.COM) file on disk.

The following example loads and saves the program DEMO.REL.

```
A>L80 DEMO,DEMO/N/E
```

The first part of the command (DEMO) loads the program called DEMO.REL. The next part (DEMO/N) saves an executable image of the program on disk in a file called DEMO.COM. The last part (/E) causes LINK-80 to exit back to CP/M. To execute the program, type DEMO at CP/M command level.

Or, you can execute the program from LINK-80 by using /G instead of /E in the command:

```
A>L80 DEMO,DEMO/N/G
```

The only difference between /G and /E is that /G executes the .COM file before exiting to CP/M.

Take note of the /N switch. It must immediately follow the filename of the .COM file you wish to save, and it does not take effect until a /E or /G switch is executed.

NOTE: If you do not specify <filename>/N after loading the program, a .COM file will not be saved.

When a /E or /G is executed, LINK-80 automatically searches the FORTRAN-80 library (FORLIB) and loads the necessary library routines.

3.3 ADDITIONAL LINK-80 FEATURES

LINK-80 also provides more specialized capabilities. For example, two or more FORTRAN programs may be loaded and linked together, and specific link/load tasks may be indicated by additional switches.

The following example links several FORTRAN object files (PART1, PART2, PART3), and saves them in a .COM file called WHOLE.COM, which is then executed.

```
A>L80 PART1,PART2,PART3,WHOLE/N/G
```

Note that the name of the .COM file must immediately precede the /N switch.

3.3.1 The Reset Switch (/R)

The /R switch returns LINK-80 to its initial state by "unloading" whatever you've loaded. Use it to reset the loader if you've made a typing mistake or loaded the wrong program. The /R switch takes effect as soon as LINK-80 sees it, so if you enter it at any time while LINK-80 is running, the loader will reset. For example:

```
A>L80
*INVEN1      (oops--meant to load INVEN2)
*/R          (reset)
*INVEN2
*            (now only INVEN2 is loaded)
```

3.3.2 The Search Switch (/S)

The /S switch is used to perform a library search. The /S switch must be preceded by the name of the library to be searched (default extension .REL).

With a multi-drive system, /S can be used to search the library if it is on a disk other than the currently logged disk.

If you have a single-drive system, /S can be used when you have your library on one diskette, and you want to save your .COM file on another diskette. This is done by first using FORLIB/S with the system library disk in the drive, then inserting another disk and entering /E or /G. (Don't forget to do /N first.) See Section 1.4.1 if you are using FORTRAN-80 with a single-drive system.

After a /S search is performed, /G or /E will not cause a library search, unless the /S left undefined globals remaining.

3.3.3 Special Switches

For typical FORTRAN-80/LINK-80 operation, only the above switches (/E, /N, /G, /R) will be needed. Some users may find that their applications require more specialized capabilities. For this reason, the following switches are also provided with LINK-80.

/O Set the typeout radix to octal. Default typeout radix is hexadecimal.

/H Return the typeout radix to hexadecimal.

/P & /D In the examples above, all programs have been loaded at the default origins for CP/M. In special cases, the user may wish to specify the origins of the programs and data that are loaded. LINK-80 provides special switches to do this.

/P and /D allow the origin(s) to be set for the NEXT program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in hexadecimal. LINK-80 does a default /P:<link origin> (i.e., 100H).

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
*/P:200
*MYPROG
Data    200     300
*/R
*/P:200/D:400
*MYPROG
Data    400     480
Program 200     280
*
```

/U List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

/M List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

/X If a <filename>/N was specified, /X will cause the file to be saved in Intel ASCII HEX format with an extension of .HEX.

Example: DEMO/N/X/E will create an Intel ASCII HEX formatted load module named DEMO.HEX.

/Y If a <filename>/N was specified, /Y will create a filename.SYM file when /E is entered. This file contains the names and addresses of all globals for use with Digital Research's Symbolic Debugger, SID and ZSID.

Example: DEMO/N/Y/E will create DEMO.COM and DEMO.SYM. DEMO/N/X/E will create DEMO.HEX and DEMO.SYM.

SUMMARY OF LINK-80 SWITCHES

/G	Execute a .REL file that has been loaded.
/E	Exit to CP/M.
<filename>/N	Create an executable object file and save it on disk as a CP/M command file. /N must be preceded by the destination filename for the .COM file.
/R	Reset LINK-80.
<library>/S	Search the library.
/O	Set the typeout radix to octal.
/H	Set the typeout radix to hexadecimal (default).
/P:<address>	Set the program origin to the specified address.
/D:<address>	Set the data origin to the specified address.
/U	List the load origin and end addresses and undefined globals.
/M	List the load origin and end addresses and all globals.
/X	Output object file in Intel HEX format.
/Y	Output object file for use with Digital Research SID and ZSID.

3.4 LINK-80 ERROR MESSAGES

LINK-80 has the following error messages:

?No Start Address	A /G switch was issued, but no main program had been loaded.
?Loading Error	The last file given for input was not a properly formatted LINK-80 object file.
?Out of Memory	Not enough memory to load program.
?Command Error	Unrecognizable LINK-80 command.
?<file> Not Found	<file>, as given in the command string, did not exist.

%2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Reorder module loading sequence or change COMMON block definitions.

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying [Program / Data] Area [,Start = xxxx / ,Public = <symbol name>(xxxx) / ,External = <symbol name>(xxxx)]

A /D or /P will cause already loaded data to be destroyed.

?Intersecting [Program / Data] Area

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

Origin [Above / Below] Loader Memory, Move Anyway (Y or N)?

While attempting to execute a /E or /G, LINK-80 discovers that either the data or program area has an origin which lies above (or below) the address that LINK-80 recognizes as the boundaries of working memory. If you answer "Y", LINK-80 will go ahead and load there, even though some other portion of the system (probably CP/M) will be destroyed. If you answer "N", LINK-80 will exit to CP/M. In either case, if a /N was given first, the COM file will already have been saved on disk.

?Can't Save Object File

A disk error occurred when the file was being saved. Usually means the disk is full.

CHAPTER 4

EXECUTING FORTRAN PROGRAMS

You may execute a FORTRAN program in one of two ways. The first is to type, at CP/M command level, the name of an executable program file (default extension .COM) that has been saved with the /N switch in the loader command string. The second is to use the /G switch in the loader command string as described in Section 3.2.

In either case, your program should execute and your output should appear on the screen. However, you may get a runtime error message. If you do, look it up in the following list, and return to the editor to debug your source file so that no error is produced at runtime.

4.1 FORTRAN-80 RUNTIME ERROR MESSAGES

The following errors may occur while a compiled program is executing. Runtime errors are surrounded by asterisks as follows:

```
**FW**
```

If the error is fatal, the hexadecimal error address is also shown as follows:

```
**FW** at address hhhh**
```

where hhhh is the hexadecimal address.

Fatal errors cause execution to cease (control is returned to the operating system). Execution continues after a warning error. However, after 20 warnings, execution ceases.

Warning Errors:

IB	Input Buffer Limit Exceeded
TL	Too Many Left Parentheses in FORMAT
OB	Output Buffer Limit Exceeded
DE	Decimal Exponent Overflow (Number in input stream had an exponent larger than 99)
IS	Integer Size Too Large
BE	Binary Exponent Overflow
IN	Input Record Too Long
OV	Arithmetic Overflow
CN	Conversion Overflow on REAL to INTEGER Conversion
SN	Argument to SIN Too Large
A2	Both Arguments of ATAN2 are 0
BI	Buffer Size Exceeded During Binary I/O
RC	Negative Repeat Count in FORMAT
FW	FORMAT Field Width is Too Small
EX	Illegal Exponentiation
GL	Computed GOTO Number Too Large
GS	Computed GOTO Number Too Small

Fatal Errors:

ID	Illegal FORMAT Descriptor
F0	FORMAT Field Width is Zero
MP	Missing Period in FORMAT
IT	I/O Transmission Error
IR	REAL Number Written to INTEGER FORMAT Field
ML	Missing Left Parenthesis in FORMAT
DZ	Division by Zero, REAL or INTEGER
LG	Illegal Argument to LOG Function (Negative or Zero)
IO	Illegal I/O Operation
SQ	Illegal Argument to SQRT Function (Negative)
DT	Data Type Does Not Agree With FORMAT Specification
EF	EOF Encountered on READ
DO	Illegal Increment or Limit in DO Loop
FN	File Not Found on OPEN
DF	Disk Full Encountered on WRITE
UN	Logical Unit Number (LUN) Too Large
OM	Out of Memory

NOTE

It is possible, in rare cases, to get a FORTRAN-80 internal error, as designated by the error code "??". This indicates an internal malfunction of the runtime. If you get the "??" error code, contact Microsoft and report the conditions under which the message appeared.

CHAPTER 5

FORTRAN-80 I/O

Most I/O is performed using FORTRAN READ and WRITE statements. The specific I/O device to be used is defined by the Logical Unit Number in the READ or WRITE statement. Logical Unit Numbers (LUNs) are used by FORTRAN to refer to an I/O device.

LUN device assignments, along with their valid I/O operations, are listed below:

LUN	DEVICE	Valid I/O Operations
1	Console	Formatted read Formatted write Unformatted write
2	Line printer	Formatted write Unformatted write Endfile
3	Console	Same as LUN #1 above. Used by the runtime system for error messages, etc., so LUN #3 should usually be used for all console I/O.
4	PUN: and RDR: (Apple slot 2)	Formatted read Formatted write Unformatted write
5	Console	Same as LUN #1 above.
6-10	Disk file (FORT06.DAT-- FORT10.DAT)	Formatted read Formatted write Unformatted read Unformatted write Backspace Endfile

Two kinds of I/O are possible: Formatted I/O and Unformatted I/O. (Unformatted I/O is also called "Binary I/O.") See FORTRAN-80 Reference Manual, Sections 8.1 and 8.2.

A formatted READ or WRITE statement will contain a reference to a FORMAT statement elsewhere in the program. The I/O is performed according to the FORMAT specification.

An unformatted READ or WRITE statement does not contain a reference to a FORMAT statement. Unformatted I/O handles data in machine readable form, rather than converting it to human readable form. Data are read or written exactly as stored in memory.

When performing formatted output of a line to a device other than a disk file, the first character of the line must be the carriage-control character. (See FORTRAN-80 Reference Manual, Section 8.7.9.) This character is not actually output, but is used to determine what action will be taken before writing the line. The carriage-control characters in FORTRAN-80 are:

Control Character	Action Taken Before Printing
0	Skip 2 lines
1	Insert form feed
+	Carriage return/no line feed
*	No action (The next WRITE will pick up where the last WRITE left off.)

NOTE: All the above carriage control characters are standard except the "*" character. This character was added to the Apple II CP/M version of FORTRAN-80.

5.1 DISK FILES

As noted above, LUNs 6 through 10 are assigned to disk files. Disk files are automatically opened when the first READ or WRITE is attempted, and they are automatically closed at the end of FORTRAN program execution.

A disk file that is OPENed by a READ or WRITE statement is given a default name that depends upon the LUN and the operating system. For CP/M, the default filenames for LUNs 6 through 10 are:

```
FORT06.DAT, FORT07.DAT,..., FORT10.DAT
```

To explicitly name a disk file, open it by calling the OPEN subroutine. (See FORTRAN-80 Reference Manual, Section 8.3.2.) Also, LUNs 1-5 may be reassigned to disk files with

the OPEN subroutine. When a LUN is assigned to a disk file with OPEN, all disk operations for LUNs 6-10 are valid.

For disk files, it is best to perform unformatted I/O. Access time is shorter, and disk space is conserved because the data are stored in machine readable form and no conversion is necessary.

5.1.1 Random Disk I/O

FORTRAN-80 supports random access I/O to disk files with one limitation: the record length is fixed at 128 bytes. If a random access WRITE does not transfer enough data to fill the record to 128 bytes, the rest of the record is filled with zeros (NUL characters). Care must be taken to insure that the data written do not exceed 128 bytes.

There is no limitation on the amount of data that can be read or written to a sequential disk file.

APPENDIX A

Transferring Files with a Single-Drive System

To transfer files from one disk to another with a single-drive system, use the following procedure:

1. Load the file to be transferred into memory using the DDT program on your CP/M system master. Just type DDT to run the DDT program. Then, when the DDT "-" prompt appears, insert the diskette containing the file to be transferred. Type I, followed by the filename of the file to be transferred, and hit RETURN. For example, to transfer the editor ED.COM, type:

   ```
   -IED.COM
   ```

   Notice that there is no space between the "I" and the filename.

2. At the "-" prompt, type R to load the file into memory. After the file is loaded, DDT will display something like:

   ```
   NEXT PC
   1B00 0100
   -
   ```

   The number under NEXT is used to determine the length of the file.

   If the last two digits under "NEXT" are 00, take the number represented by the first two digits and subtract one. For example, 1B minus 01 is 1A. Convert this number to its decimal equivalent: 1A is 1*16+10=26.

   If the last two digits of the number under "NEXT" are 80, take the number represented by the first two digits and convert this number to its decimal equivalent. Don't subtract one if the last two digits are 80.

The resulting number is the length of the file divided by 256.

Type control-C to return to CP/M command level.

3. Insert the diskette you want to transfer the file to, and type control-C. (Don't forget to type control-C AFTER changing disks.) Use the CP/M SAVE command to save the file to the disk, using the number obtained above. For example:

```
SAVE 26 ED.COM
```

When the CP/M prompt reappears, the transfer is complete.

APPENDIX B

Programs in ROM

If a FORTRAN program is intended for ROM, the programmer should be aware of the following ramifications:

1. The /M switch must be used at (F80) compile time. When a /M is specified, the generated code will differ from normal in the following ways:

    1. FORMATs will be placed in the program area, with a "JMP" around them.

    2. Parameter blocks (for sub- program calls with more than 3 parameters) will be initialized at runtime, rather than being initialized by the loader.

2. DATA statements should not be used to initialize RAM. Such initialization is done by the loader, and will therefore not be present at execution. Variables and arrays may be initialized during execution via assignment statements, or by READing into them.

3. FORMATs should not be read into during execution.

4. If the standard library I/O routines are used, DISK files cannot be OPENed. Therefore, LUNs 6-10 must be used. If you wish to change default filenames, etc., you must modify and reassemble DSKDRV.MAC.

5. A library routine, $INIT, sets the stack pointer at the top of available memory (as indicated by the operating system) before execution begins. This is the standard FORTRAN-80 initialization routine.

The calling convention is:

```
LXI     B,<return address>
JMP     $INIT
```

If the generated code is intended for some other machine, INIT.MAC may have to be modified and reassembled. The source of INIT.MAC is provided on the FORTRAN-80 disk. Only the portion of this routine which sets up the stack pointer should ever be modified by the user.

NOTE: To modify and reassemble any assembly language source file (such as DSKDRV.MAC or INIT.MAC), you must use the Microsoft MACRO-80 Assembler. See the FORTRAN-80 Reference Manual, Appendix B. (MACRO-80 is supplied with the Assembly Language Development System, available from Microsoft Consumer Products.)

fortran-80 reference manual

Microsoft FORTRAN-80
for the Apple II Computer

* * *

FORTRAN-80 for the Apple II Computer has many new features that take advantage of the Apple II hardware and the CP/M operating system. These features are described in Appendix A of this manual.

* * *

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

It is against the law to copy any of the Microsoft FORTRAN-80 package on cassette tape, disk or any other medium for any purpose other than personal convenience.

Copyright Microsoft, 1980

LIMITED WARRANTY

MICROSOFT shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

CP/M is a registered trademark of Digital Research Inc.

Apple is a registered trademark of Apple Computer Inc.

SoftCard is a trademark of Microsoft

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

8211-342-02

MICROSOFT FORTRAN-80
Reference Manual

Contents

SECTION 1 Introduction 7

SECTION 2 Fortran Program Form 8

2.1 Fortran Line Format 8
2.2 Statement Label 10
2.3 Statements 11
2.4 INCLUDE Statement 12

SECTION 3 Data Representation/Storage Format . . 13

3.1 Data Names and Types 13
3.1.1 Names 13
3.1.2 Types 13
3.2 Constants 14
3.3 Variables 18
3.4 Arrays and Array Elements 19
3.5 Subscripts 20
3.6 Data Storage Allocation 21

SECTION 4 FORTRAN Expressions 24

4.1 Arithmetic Expressions 24
4.2 Expression Evaluation 25
4.3 Logical Expressions 26
4.3.1 Relational Expressions 27
4.3.2 Logical Operators 28
4.4 Hollerith, Literal, and Hexadecimal Constants in Expression 30

SECTION 5 Replacement Statements 31

SECTION 6 Specification Statements 33

6.1 Specification Statements 33
6.2 Array Declarators 33
6.3 Type Statements 34
6.4 EXTERNAL Statements 36
6.5 DIMENSION Statements 36
6.6 COMMON Statements 37

6.7 EQUIVALENCE Statements 38
6.8 DATA Initialization Statement 40
6.9 IMPLICIT Statement 42

SECTION 7 FORTRAN Control Statements 43

7.1 GOTO Statements 44
7.1.1 Unconditional GOTO 44
7.1.2 Computed GOTO 45
7.1.3 Assigned GOTO 46
7.2 ASSIGN Statement 47
7.3 IF Statement 48
7.3.1 Arithmetic IF 48
7.3.2 Logical IF 49
7.4 DO Statement 50
7.5 CONTINUE Statement 53
7.6 STOP Statement 54
7.7 PAUSE Statement 54
7.8 CALL Statement 55
7.9 RETURN Statement 55
7.10 END Statement 55

SECTION 8 Input/Output 56

8.1 Formatted READ/WRITE 57
8.1.1 Formatted READ 57
8.1.2 Formatted WRITE Statements 60
8.2 Unformatted READ/WRITE 61
8.3 Disk File I/O 62
8.3.1 Random Disk I/O 63
8.3.2 OPEN Subroutine 64
8.4 Auxiliary I/O Statements 65
8.5 ENCODE/DECODE 65
8.6 Input/Output List Specifications . . . 66
8.6.1 List Item Types 66
8.6.2 Special Notes on List Specifications 68
8.7 FORMAT Statements 69
8.7.1 Field Descriptors 70
8.7.2 Numeric Conversions 71
8.7.3 Hollerith Conversions 75
8.7.4 Logical Conversion 77
8.7.5 X Descriptor 78
8.7.6 P Descriptor 79
8.7.7 Special Control Features of FORMAT Statements 80
8.7.7.1 Repeat Specifications 80
8.7.7.2 Field Separators 81
8.7.8 FORMAT Control, List Specifications, and Record Demarcation 82
8.7.9 FORMAT Carriage Control 84
8.7.10 FORMAT Specifications in Arrays . . 84

SECTION 9 Functions and Subprograms 86

9.1 PROGRAM Statement 87
9.2 Statement Functions 87
9.3 Library Functions 88
9.4 Function Subprograms 92
9.5 Construction of Function Subprograms . 92
9.6 Referencing a Function Subprogram . . 94
9.7 Subroutine Subprograms 95
9.8 Construction of Subroutine Subprograms 95
9.9 Referencing a Subroutine Subprogram . 97
9.10 Return From Function and Subroutine Subprograms 98
9.11 Processing Arrays in Subprograms . . . 99
9.12 BLOCK DATA Subroutine 100
9.13 Program Chaining 101

APPENDIX A- FORTRAN-80 Language Extensions including New Apple II Features . . .103

APPENDIX B- I/O Interface 111

APPENDIX C- Subprogram Linkages 113

APPENDIX D- ASCII Character Codes 116

APPENDIX E- Referencing FORTRAN-80 Library Subroutines 117

SECTION 1

INTRODUCTION

FORTRAN is a universal, problem oriented programming language designed to simplify the preparation and check-out of computer programs. The name of the language - FORTRAN - is an acronym for FORmula TRANslator.

The syntactical rules for using the language are rigorous and require the programmer to define fully the characteristics of a problem in a series of precise statements. These statements, called the source program, are translated by a system program called the FORTRAN processor into an object program in the machine language of the computer on which the program is to be executed.

This manual defines the FORTRAN source language for the 8080 and Z-80 microcomputers. This language includes the American National Standard FORTRAN language as described in ANSI document X3.9-1966, approved on March 7, 1966, plus a number of language extensions and some restrictions. These language extensions and restrictions are described in the text of this document and are listed in Appendix A.

NOTE

This FORTRAN differs from the Standard in that it does not include the COMPLEX data type.

Examples are included throughout the manual to illustrate the construction and use of the language elements. The programmer should be familiar with all aspects of the language to take full advantage of its capabilities.

Section 2 describes the form and components of an FORTRAN-80 source program. Sections 3 and 4 define data types and their expressional relationships. Sections 5 through 9 describe the proper construction and usage of the various statement classes.

SECTION 2

FORTRAN PROGRAM FORM

FORTRAN-80 source programs consist of one program unit called the Main program and any number of program units called subprograms. A discussion of subprogram types and methods of writing and using them is in Section 9 of this manual.

Programs and program units are constructed of an ordered set of statements which precisely describe procedures for solving problems and which also define information to be used by the FORTRAN processor during compilation of the object program. Each statement is written using the FORTRAN character set and following a prescribed line format.

2.2 FORTRAN LINE FORMAT

A FORTRAN program line consists of up to 80 columns, divided into four fields:

1. Statement Label (or Number) field- Columns 1 through 5 (See definition of statement labels).

2. Continuation character field-
   Column 6

3. Statement field-
   Columns 7 through 72

4. Indentification field-
   Columns 73 through 80

The identification field is available for any purpose the FORTRAN programmer may desire and is ignored by the FORTRAN processor.

The lines of a FORTRAN statement are placed in Columns 1 through 72 formatted according to line types. The four line types, their definitions, and column formats are:

1. Initial Line -- the first or only line of each statement.

   1. Columns 1-5 may contain a statement label to identify the statement.

2. Column 6 must be blank.

3. Columns 7-72 contain all or part of the statement.

4. An initial line may begin anywhere within the statement field.

Example:

```
C THE STATEMENT BELOW CONSISTS
C    OF AN INITIAL LINE
C
          A= .5*SQRT(3-2.*C)
```

2. Continuation Line -- used when additional lines of coding are required to complete a statement originating with an initial line.

1. Columns 1-5 are ignored, unless Column 1 contains a C.

2. If Column 1 contains a C, it is a comment line.

3. Column 6 must contain a character other than zero or blank.

4. Columns 7-72 contain the continuation of the statement.

5. There may be as many continuation lines as needed to complete the statement.

Example:

```
C  THE STATEMENTS BELOW ARE AN INITIAL LINE
C       AND 2 CONTINUATION LINES
C
  63  BETA(1,2) =
     1     A6BAR**7-(BETA(2,2)-A5BAR*50
     2     +SQRT (BETA(2,1)))
```

3. Comment line -- used for source program annotation at the convenience of the programmer.

1. Column 1 contains the letter C.

2. Columns 2 - 72 are used in any desired format to express the comment or they may be left blank.

3. A comment line may be followed only by an initial line, an END line, or another comment line.

4. Comment lines have no effect on the object program and are ignored by the FORTRAN processor except for display purposes in the listing of the program.

   Example:

```
C     COMMENT LINES ARE INDICATED BY THE
C        CHARACTER C IN COLUMN 1.
C  THESE ARE COMMENT LINES
```

4. END line -- the last line of a program unit.

   1. Columns 1-5 may contain a statement label.

   2. Column 6 must be blank.

   3. Columns 7-72 contain the END statement.

   4. Each FORTRAN program unit must have an END line as its last line to inform the Processor that it is at the physical end of the program unit.

   5. An END line may follow any other type line.

2.2 STATEMENT LABEL

A statement label may be placed in columns 1-5 of a FORTRAN statement initial line and is used for reference purposes in other statements.

The following considerations govern the use of statement labels:

1. The label is an integer from 1 to 99999.

2. The numeric value of the label, leading zeros and blanks are not significant.

3. A label must be unique within a program unit.

4. A label on a continuation line is ignored by the FORTRAN Processor.

Example:

```
C  EXAMPLES OF STATEMENT LABELS
C
  1
  101
99999
 763
```

2.3 STATEMENTS

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or non-executable.

Executable statements specify actions and cause the FORTRAN Processor to generate object program instructions. There are three types of executable statements:

1. Replacement statements.
2. Control statements.
3. Input/Output statements.

Non-executable statements describe to the processor the nature and arrangement of data and provide information about input/output formats and data initialization to the object program during program loading and execution. There are five types of non-executable statements:

1. Specification statements.
2. DATA Initialization statements.
3. FORMAT statements.
4. FUNCTION defining statements.
5. Subprogram statements.

The proper usage and construction of the various types of statements are described in Sections 5 through 9.

2.4 INCLUDE STATEMENT

The INCLUDE statement causes the compiler to bring an outside FORTRAN source program into the current program. The format of the statement is

INCLUDE<filename>

Use of INCLUDE eliminates the need to repeat an often-used sequence of statements in the current source file.

SECTION 3

DATA REPRESENTATION / STORAGE FORMAT

The FORTRAN Language prescribes a definitive method for identifying data used in FORTRAN programs by name and type.

3.1 DATA NAMES AND TYPES

3.1.1 NAMES

1. Constant - An explicitly stated datum.

2. Variable - A symbolically identified datum.

3. Array - An ordered set of data in 1, 2 or 3 dimensions.

4. Array Element - One member of the set of data of an array.

3.1.2 TYPES

1. Integer -- Precise representation of integral numbers (positive, negative or zero) having precision to 5 digits in the range -32768 to +32767 inclusive (-2**15 to 2**15-1).

2. Real -- Approximations of real numbers (positive, negative or zero) represented in computer storage in 4-byte, floating-point form. Real data are precise to 7+ significant digits and their magnitude may lie between the approximate limits of 10**-38 and 10**38 (2**-127 and 2**127).

3. Double Precision -- Approximations of real numbers (positive, negative or zero) represented in computer storage in 8-byte, floating-point form. Double Precision data are precise to 16+ significant digits in the same magnitude range as real data.

4. Logical -- One byte representations of the truth values "TRUE" or "FALSE" with "FALSE defined to have an internal representation of zero. The constant .TRUE. has the value -1, however any non-zero value will be treated as .TRUE. in a Logical IF statement. In addition, Logical types may be used as one byte signed integers in the range -128 to +127, inclusive.

5. Hollerith -- A string of any number of characters from the computer's character set. All characters including blanks are significant. Hollerith data require one byte for storage of each character in the string.

6. Extended Integer -- INTEGER*4 is an extended precision representation using 32-bit two's complement (four 8-bit bytes) for 9+ significant digits in the range -2147483648 to +2147483647 inclusive (-2**31 to 2**31-1).

   Integer*4 variables cannot be used as Logical Unit Numbers, array indices, implied DO loop indices, or as the control variable in computed or assigned GOTO statements.

3.2 CONSTANTS

FORTRAN constants are identified explicitly by stating their actual value. The plus (+) character need not precede positive valued constants.

Formats for writing constants are shown in Table 3-1.

Table 3-1. CONSTANT FORMATS

TYPE	FORMATS AND RULES OF USE	EXAMPLES
INTEGER	1. 1 to 5 decimal digits interpreted as a decimal number.	-763 1 +00672
	2. A preceding plus (+) or minus (-) sign is optional.	-32768 +32767
	3. No decimal point (.) or comma (,) is allowed but spaces are permitted in the source program.	- 32 767
	4. Value range: -32768 through +32767 (.i.e., -2**15 through 2**15-1).	
REAL	1. A decimal number with precision to 7 digits and represented in one of the following forms: a. + or -.f + or -i.f b. + or -i.E+ or -e + or -.fE+ or -e + or -i.fE+ or -e where i, f, and e are each strings representing integer, fraction, and exponent respectively.	345. -.345678 +345.678 +.3E3 -73E4
	2. Plus (+) and minus (-) characters are optional.	
	3. In the form shown in 1 b above, if r represents any of the forms preceding E+ or -e (i.e., rE+ or -e), the value of the constant is interpreted as r times 10**e, where -38=E=38.	

TYPE	FORMATS AND RULES OF USE	EXAMPLES
	4. If the constant preceding E+ or -e contains more significant digits than the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.	
DOUBLE PRECISION	A decimal number with precision to 16 digits. All formats and rules are identical to those for REAL constants, except D is used in place of E. Note that a real constant is assumed single precision unless it contains a "D" exponent.	+345.678 +.3D3 -73D4
LOGICAL	.TRUE. generates a non-zero byte (hexadecimal FF) and .FALSE. generates a byte in which all bits are 0. If logical values are used as one-byte integers, the rules for use are the same as for type INTEGER, except that the range allowed is -128 to +127, inclusive.	.TRUE. .FALSE.

TYPE	FORMATS AND RULES OF USE	EXAMPLES
LITERAL	In the literal form, any number of characters may be enclosed by single quotation marks. The form is as follows: 'X1X2X3...Xn' where each Xi is any character other than '. Two quotation marks in succession may be used to represent the quotation mark character within the string, i.e., if X2 is to be the quotation mark character, the string appears as the following: 'X1''X3...Xn'	
HEXADECIMAL	1. The letter Z or X followed by a single quote, up to 4 hexadecimal digits (0-9 and A-F) and a single quote is recognized as a hexadecimal value. 2. A hexadecimal constant is right justified in its storage value.	Z'12' X'AB1F' Z'FFFF' X'1F'
INTEGER*4	1. One to ten decimal digits interpreted as a decimal number. 2. A preceding plus or minus sign is optional. 3. No deciml point or comma is allowed, but spaces are permitted in the source program. 4. Value range: -2147483648 through 2147483647 (i.e., -2**31 through 2**31-1).	1234567890 0 -2147483647 - 2 147 483 647

3.3 VARIABLES

Variable data are identified in FORTRAN statements by symbolic names. The names are unique strings of from 1 to 6 alphanumeric characters of which the first is a letter.

NOTE

System variable names and runtime subprogram names are distinguished from other variable names in that they begin with the dollar sign character ($). It is therefore strongly recommended that in order to avoid conflicts, symbolic names in FORTRAN source programs begin with some letter other than "$".

Examples:

I5, TBAR, B23, ARRAY, XFM79, MAX, A1$C

Variable data are classified into four types: INTEGER, REAL, DOUBLE PRECISION and LOGICAL. The specification of type is accomplished in one of the following ways:

1. Implicit typing in which the first letter of the symbolic name specifies Integer or Real type. Unless explicitly typed (2., below), symbolic names beginning with I, J, K, L, M or N represent Integer variables, and symbolic names beginning with letters other than I, J, K, L, M or N represent Real variables.

   Integer Variables

   ITEM
   J1
   MODE
   K123
   N2

Real Variables

BETA
H2
ZAP
AMAT
XID

2. Variables may be typed explicitly. That is, they may be given a particular type without reference to the first letters of their names. Variables may be explicitly typed as INTEGER, REAL, DOUBLE PRECISION or LOGICAL. The specific statements used in explicitly typing data are described in Section 6.

Variable data receive their numeric value assignments during program execution or, initially, in a DATA statement (Section 6).

Hollerith or Literal data may be assigned to any type variable. Sub-paragraph 3.6 contains a discussion of Hollerith data storage.

3.4 ARRAYS AND ARRAY ELEMENTS

An array is an ordered set of data characterized by the property of dimension. An array may have 1, 2 or 3 dimensions and is identified and typed by a symbolic name in the same manner as a variable except that an array name must be so declared by an "array declarator." Complete discussions of the array declarators appear in Section 6 of this manual. An array declarator also indicates the dimensionality and size of the array. An array element is one member of the data set that makes up an array. Reference to an array element in a FORTRAN statement is made by appending a subscript to the array name. The term array element is synonymous with the term subscripted variable used in some FORTRAN texts and reference manuals.

An initial value may be assigned to any array element by a DATA statement or its value may be derived and defined during program execution.

3.5 SUBSCRIPTS

A subscript follows an array name to uniquely identify an array element. In use, a subscript in a FORTRAN statement takes on the same representational meaning as a subscript in familiar algebraic notation.

Rules that govern the use of subscripts are as follows:

1. A subscript contains 1, 2 or 3 subscript expressions (see 4 below) enclosed in parentheses.

2. If there are two or three subscript expressions within the parentheses, they must be separated by commas.

3. The number of subscript expressions must be the same as the specified dimensionality of the Array Declarator except in EQUIVALENCE statements (Section 6).

4. A subscript expression is written in one of the following forms:

   K C*V V-K
   V C*V+K C*V-K
   V+K

   where C and K are integer constants and V is an integer variable name (see Section 4 for a discussion of expression evaluation).

5. Subscripts themselves may not be subscripted. Examples:

   X(2*J-3,7) A(I,J,K) I(20) C(L-2) Y(I)

3.6 DATA STORAGE ALLOCATION

Allocation of storage for FORTRAN data is made in numbers of storage units. A storage unit is the memory space required to store one real data value (4 bytes).

Table 3-2 defines the word formats of the three data types.

Hexadecimal data may be associated (via a DATA statement) with any type data. Its storage allocation is the same as the associated datum.

Hollerith or literal data may be associated with any data type by use of DATA initializaton statements (Section 6).

Up to eight Hollerith characters may be associated with Double Precision type storage, up to four with Real or Integer*4, up to two with Integer*2, and one with Logical type storage.

TABLE 3-2. STORAGE ALLOCATION BY DATA TYPES

TYPE	ALLOCATION
INTEGER	2 bytes/ 1/2 storage unit

S	Binary Value

Negative numbers are the two's complement of positive representations. The storage order is reversed. The least significant byte is followed by the most significant byte.

LOGICAL — 1 byte/ 1/4 storage unit

Zero	(false) or	non-zero	(true)

A non-zero valued byte indicates true (the logical constant .TRUE. is represented by the hexadecimal value FF). A zero valued byte indicates false.

When used as an arithmetic value, a Logical datum is treated as an Integer in the range -128 to +127.

REAL — 4 bytes/ 1 storage unit

Characteristic	S	Mantissa (hi)
Mantissa(mid)		Mantissa (low)

The first byte is the characteristic expressed in excess 200 (octal) notation; i.e., a value of 200 (octal) corresponds to a binary exponent of 0. Values less than 200 (octal) correspond to negative exponents, and values greater than 200 correspond to positive exponents. By definition, if the characteristic is zero, the entire number is zero.

The next three bytes constitute the mantissa. The mantissa is always normalized such that the high order bit is one, eliminating the need to actually save that bit. The high bit is used instead to indicate the sign of the number. A one indicates a negative number, and zero indicates a positive number. The mantissa is assumed to be a binary fraction whose binary point is to the left of the mantissa. The format of the mantissa is "signed magnitude." The bytes are stored in

reverse order: mantissa low order, followed by mid order, high order, and characteristic.

DOUBLE PRECISION

8 bytes/ 2 storage units

The internal form of Double Precision data is identical with that of Real data except Double Precision uses 4 extra bytes for the matissa.

INTEGER*4

4 bytes/ 1 storage unit

Negative numbers are represented in two's complement form. The bytes are stored in reverse order, least significant to most significant.

TABLE 3-3. EQUIVALENT DATA TYPES AND SIZES

Equivalent Representations	Size in Bytes
BYTE	1
INTEGER*1	1
LOGICAL	1
INTEGER	2
INTEGER*2	2
LOGICAL*2	2
INTEGER*4	4
LOGICAL*4	4
REAL	4
REAL*4	4
DOUBLE PRECISION	8
REAL*8	8

SECTION 4

FORTRAN EXPRESSIONS

A FORTRAN expression is composed of a single operand or a string of operands connected by operators. Two expression types --Arithmetic and Logical-- are provided by FORTRAN. The operands, operators and rules of use for both types are described in the following paragraphs.

4.1 ARITHMETIC EXPRESSIONS

The following rules define all permissible arithmetic expression forms:

1. A constant, variable name, array element reference or FUNCTION reference (Section 9) standing alone is an expression.

   Examples:

   S(I) JOBNO 217 17.26 SQRT(A+B)

2. If E is an expression whose first character is not an operator, then +E and -E are called signed expressions.

   Examples:

   -S +JOBNO -217 +17.26 -SQRT(A+B)

3. If E is an expression, then (E) means the quantity resulting when E is evaluated.

   Examples:

   (-A) -(JOBNO) -(X+1) (A-SQRT(A+B))

4. If E is an unsigned expression and F is any expression, then: F+E, F-E, F*E, F/E and F**E are all expressions.

   Examples:

   ```
   -(B(I,J)+SQRT(A+B(K,L)))
   1.7E-2**(X+5.0)
   -(B(I+3,3*J+5)+A)
   ```

5. An evaluated expression may be Integer, Extended Integer, Real, Double Precision, or Logical. The type is determined by the data types of the elements of the expression. If

the elements of the expression are not all of the same type, the type of the expression is determined by the element having the highest type. The type hierarchy (highest to lowest) is as follows: DOUBLE PRECISION, REAL, INTEGER*4, INTEGER, LOGICAL.

6. Expressions may contain nested parenthesized elements as in the following:

```
A*(Z-((Y+X)/T))**J
```

where Y+X is the innermost element, (Y+X)/T is the next innermost, Z-((Y+X)/T) the next. In such expressions, care should be taken to see that the number of left parentheses and the number of right parentheses are equal.

4.2 EXPRESSION EVALUATION

Arithmetic expressions are evaluated according to the following rules:

1. Parenthesized expression elements are evaluated first. If parenthesized elements are nested, the innermost elements are evaluated, then the next innermost until the entire expression has been evaluated.

2. Within parentheses and/or wherever parentheses do not govern the order or evaluation, the hierarchy of operations in order of precedence is as follows:

   a. FUNCTION evaluation
   b. Exponentiation
   c. Multiplication and Division
   d. Addition and Subtraction

Example:

The expression

```
A*(Z-((Y+R)/T))**J+VAL
```

is evaluated in the following sequence:

```
e1 = Y+R
e2 = (e1)/T
e3 = Z-e2
e4 = e3**J
e5 = A*e4
e6 = e5+VAL
```

3. The expression X**Y**Z is not allowed. It should be written as follows:

   (X**Y)**Z or X**(Y**Z)

4. Use of an array element reference requires the evaluation of its subscript. Subscript expressions are evaluated under the same rules as other expressions.

4.3 LOGICAL EXPRESSIONS

A Logical Expression may be any of the following:

1. A single Logical Constant (i.e., .TRUE. or .FALSE.), a Logical variable, Logical Array Element or Logical FUNCTION reference (see FUNCTION, Section 9).

2. Two arithmetic expressions separated by a relational operator (i.e., a relational expression).

3. Logical operators acting upon logical constants, logical variables, logical array elements, logical FUNCTIONS, relational expressions or other logical expressions.

The value of a logical expression is always either .TRUE. or .FALSE.

4.3.1 RELATIONAL EXPRESSIONS

The general form of a relational expression is as follows:

```
el r e2
```

where el and e2 are arithmetic expressions and r is a relational operator. The six relational operators are as follows:

.LT.	Less Than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The value of the relational expression is .TRUE. if the condition defined by the operator is met. Otherwise, the value is .FALSE.

Examples:

```
A.EQ.B
(A**J).GT.(ZAP*(RHO*TAU-ALPH))
```

4.3.2 LOGICAL OPERATORS

Table 4-1 lists the logical operations. U and V denote logical expressions.

Table 4-1. Logical Operations

.NOT.U	The value of this expression is the logical complement of U (i.e., 1 bits become 0 and 0 bits become 1).
U.AND.V	The value of this expression is the logical product of U and V (i.e., there is a 1 bit in the result only where the corresponding bits in both U and V are 1.
U.OR.V	The value of this expression is the logical sum of U and V (i.e., there is a 1 in the result if the corresponding bit in U or V is 1 or if the corresponding bits in both U and V are 1.
U.XOR.V	The value of this expression is the exclusive OR of U and V (i.e., there is a one in the result if the corresponding bits in U and V are 1 and 0 or 0 and 1 respectively.

Examples:

If U = 01101100 and V = 11001001 , then

```
.NOT.U  = 10010011
U.AND.V = 01001000
U.OR.V  = 11101101
U.XOR.V = 10100101
```

The following are additional considerations for construction of Logical expressions:

1. Any Logical expression may be enclosed in parentheses. However, a Logical expression to which the .NOT. operator is applied must be enclosed in parentheses if it contains two or more elements.

2. In the hierarchy of operations, parentheses may be used to specify the ordering of the expression evaluation. Within parentheses, and where parentheses do not dictate evaluation order, the order is understood to be as follows:

   a. FUNCTION Reference
   b. Exponentiation (**)
   c. Multiplication and Division (* and /)
   d. Addition and Subtraction (+ and -)
   e. .LT., .LE., .EQ., .NE., .GT., .GE.
   f. .NOT.
   g. .AND.
   h. .OR., .XOR.

   Examples:

   The expression

```
X .AND. Y .OR. B(3,2) .GT. Z
```

   is evaluated as

```
e1 = B(3,2).GT.Z
e2 = X .AND. Y
e3 = e2 .OR. e1
```

   The expression

```
X .AND. (Y .OR. B(3,2) .GT. Z)
```

   is evaluated as

```
e1 = B(3,2) .GT. Z
e2 = Y .OR. e1
e3 = X .AND. e2
```

3. It is invalid to have two contiguous logical operators except when the second operator is .NOT. That is,

   .AND..NOT.

   and

   .OR..NOT.

   are permitted.

   Example:

   A.AND..NOT.B is permitted

   A.AND..OR.B is not permitted

4.4 HOLLERITH, LITERAL, AND HEXADECIMAL CONSTANTS IN EXPRESSIONS

Hollerith, Literal, and Hexadecimal constants are allowed in expressions in place of Integer constants. These special constants always evaluate to an Integer value and are therefore limited to a length of two bytes. The only exceptions to this are:

1. Long Hollerith or Literal constants may be used as subprogram parameters.

2. Hollerith, Literal, or Hexadecimal constants may be up to four bytes long in DATA statements when associated with Real variables, or up to eight bytes long when associated with Double Precision variables.

SECTION 5

REPLACEMENT STATEMENTS

Replacement statements define computations and are used similarly to equations in normal mathematical notation. They are of the following form:

```
v = e
```

where v is any variable or array element and e is an expression.

FORTRAN semantics defines the equality sign (=) as meaning to be replaced by rather than the normal is equivalent to. Thus, the object program instructions generated by a replacement statement will, when executed, evaluate the expression on the right of the equality sign and place that result in the storage space allocated to the variable or array element on the left of the equality sign.

The following conditions apply to replacement statements:

1. Both v and the equality sign must appear on the same line. This holds even when the statement is part of a logical IF statement (section 7).

   Example:

```
C IN A REPLACEMENT STATEMENT THE '='
C     MUST BE IN THE INITIAL LINE.
      A(5,3) =
     1     B(7,2) + SIN(C)
```

   The line containing v= must be the initial line of the statement unless the statement is part of a logical IF statement. In that case the v= must occur no later than the end of the first line after the end of the IF.

2. If the data types of the variable, v, and the expression, e, are different, then the value determined by the expression will be converted, if possible, to conform to the typing of the variable. Table 5-1 shows which type expressions may be equated to which type of variable. Y indicates a valid replacement and N indicates an invalid replacement. Footnotes to Y indicate conversion considerations.

Table 5-1. Replacement By Type

	Expression Types (e)				
Variable Types	Integer	Real	Logical	Double	Ext Int
Integer	Y	Ya	Yb	Ya	Yg
Real	Yc	Y	Yc	Ye	Yc
Logical	Yd	Ya	Y	Ya	Yd
Double	Yc	Y	Yc	Y	Yc
Ext Int	Yf	Yh	Yb,f	Yh	Y

a. The Real expression value is converted to Integer, truncated if necessary to conform to the range of Integer data.
b. The sign is extended through the second byte.
c. The variable is assigned the Real approximation of the Integer value of the expression.
d. The variable is assigned the truncated value of the Integer expression (the low-order byte is used, regardless of sign).
e. The variable is assigned the rounded value of the Real expression.
f. The sign is extended through the third and fourth bytes.
g. The variable is assigned the truncated value of the Extended Integer expression.
h. The expression value is converted to Extended Integer and truncated to conform to the range of Extended Integer data.

SECTION 6

SPECIFICATION STATEMENTS

Specification statements are non-executable, non-generative statements which define data types of variables and arrays, specify array dimensionality and size, allocate data storage or otherwise supply determinative information to the FORTRAN processor. DATA intialization statements are non-executable, but generate object program data and establish initial values for variable data.

6.1 SPECIFICATION STATEMENTS

There are seven kinds of specification statements. They are as follows:

IMPLICIT statements
Type, EXTERNAL, and DIMENSION statements
COMMON statements
EQUIVALENCE statements
DATA initialization statements

All specification statements are grouped at the beginning of a program unit and must be ordered as they appear above. Specification statements may be preceded only by a FUNCTION, SUBROUTINE, PROGRAM or BLOCK DATA statement. All specification statements must precede statement functions and the first executable statement.

6.2 ARRAY DECLARATORS

Three kinds of specification statements may specify array declarators. These statements are the following:

Type statements
DIMENSION statements
COMMON statements

Of these, DIMENSION statements have the declaration of arrays as their sole function. The other two serve dual purposes. These statements are defined in subparagraphs 6.3, 6.5 and 6.6.

Array declarators are used to specify the name, dimensionality and sizes of arrays. An array may be declared only once in a program unit.

An array declarator has one of the following forms:

```
ui (k)
ui (k1,k2)
ui (k1,k2,k3)
```

where ui is the name of the array, called the declarator name, and the k's are integer constants.

Array storage allocation is established upon appearance of the array declarator. Such storage is allocated linearly by the FORTRAN processor where the order of ascendancy is determined by the first subscript varying most rapidly and the last subscript varying least rapidly.

For example, if the array declarator AMAT(3,2,2) appears, storage is allocated for the 12 elements in the following order:

AMAT(1,1,1), AMAT(2,1,1), AMAT(3,1,1), AMAT(1,2,1), AMAT(2,2,1), AMAT(3,2,1), AMAT(1,1,2), AMAT(2,1,2), AMAT(3,1,2), AMAT(1,2,2), AMAT(2,2,2), AMAT(3,2,2)

6.3 TYPE STATEMENTS

Variable, array and FUNCTION names are automatically typed Integer or Real by the 'predefined' convention unless they are changed by Type statements. For example, the type is Integer if the first letter of an item is I, J, K, L, M or N. Otherwise, the type is Real.

Type statements provide for overriding or confirming the pre-defined convention by specifying the type of an item. In addition, these statements may be used to declare arrays.

Type statements have the following general form:

```
t vl,v2,...vn
```

where t represents one of the terms INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4, or BYTE. Each v is an array declarator or a variable, array or FUNCTION name. The INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, REAL*8, LOGICAL*1, LOGICAL*2, LOGICAL*4 types are allowed for readability and compatibility with other FORTRANs.

Example:

```
REAL AMAT(3,3,5),BX,IETA,KLPH
```

NOTE

1. AMAT and BX are redundantly typed.
2. IETA and KLPH are unconditionally declared Real.
3. AMAT(3,3,5) is a constant array declarator specifying an array of 45 elements.

Example:

```
INTEGER M1, HT, JMP(15), FL
```

NOTE

M1 is redundantly typed here. Typing of HT and FL by the pre-defined convention is overridden by their appearance in the INTEGER statement. JMP(15) is a constant array declarator. It redundantly types the array elements as Integer and communicates to the processor the storage requirements and dimensionality of the array.

Example:

```
LOGICAL L1, TEMP
```

NOTE

All variables, arrays or FUNCTIONs required to be typed Logical must appear in a LOGICAL statement, since no starting letter indicates these types by the default convention.

6.4 EXTERNAL STATEMENTS

EXTERNAL statements have the following form:

```
EXTERNAL ul,u2,...,un
```

where each ui is a SUBROUTINE, BLOCK DATA or FUNCTION name. When the name of a subprogram is used as an argument in a subprogram reference, it must have appeared in a preceding EXTERNAL statement.

When a BLOCK DATA subprogram is to be included in a program load, its name must have appeared in an EXTERNAL statement within the main program unit.

For example, if SUM and AFUNC are subprogram names to be used as arguments in the subroutine SUBR, the following statements would appear in the calling program unit:

```
.
.
.
EXTERNAL SUM, AFUNC
.
.
.
.
CALL SUBR(SUM,AFUNC,X,Y)
```

6.5 DIMENSION STATEMENTS

A DIMENSION statement has the following form:

```
DIMENSION u2,u2,u3,...,un
```

where each ui is an array declarator.

Example:

```
DIMENSION RAT(5,5),BAR(20)
```

This statement declares two arrays - the 25 element array RAT and the 20 element array BAR.

6.6 COMMON STATEMENTS

COMMON statements are non-executable, storage allocating statements which assign variables and arrays to a storage area called COMMON storage and provide the facility for various program units to share the use of the same storage area.

COMMON statements are expressed in the following form:

```
COMMON /y1/a1/y2/a2/.../yn/an
```

where each yi is a COMMON block storage name and each ai is a sequence of variable names, array names or constant array declarators, separated by commas. The elements in ai make up the COMMON block storage area specified by the name yi. If any yi is omitted leaving two consecutive slash characters (//), the block of storage so indicated is called blank COMMON. If the first block name (y1) is omitted, the two slashes may be omitted.

Example:

```
      COMMON /AREA/A,B,C/BDATA/X,Y,Z,
     X                FL,ZAP(30)
```

In this example, two blocks of COMMON storage are allocated - AREA with space for three variables and BDATA, with space for four variables and the 30 element array, ZAP.

Example:

```
      COMMON //A1,B1/CDATA/ZOT(3,3)
     X            //T2,Z3
```

In this example, A1, B1, T2 and Z3 are assigned to blank COMMON in that order. The pair of slashes preceding A1 could have been omitted.

CDATA names COMMON block storage for the nine element array, ZOT and thus ZOT (3,3) is an array declarator. ZOT must not have been previously declared. (See "Array Declarators," Paragraph 6.3.)

Additional Considerations:

1. The name of a COMMON block may appear more than once in the same COMMON statement, or in more than one COMMON statement.

2. A COMMON block name is made up of from 1 to 6 alphanumeric characters, the first of which must be a letter.

3. A COMMON block name must be different from any subprogram names used throughout the program.

4. The size of a COMMON area may be increased by the use of EQUIVALENCE statements. See "EQUIVALENCE Statements," Paragraph 6.7.

5. The lengths of COMMON blocks of the same name need not be identical in all program units where the name appears. However, if the lengths differ, the program unit specifying the greatest length must be loaded first (see the discussion of LINK-80 in the User's Guide). The length of a COMMON area is the number of storage units required to contain the variables and arrays declared in the COMMON statement (or statements) unless expanded by the use of EQUIVALENCE statements.

6.7 EQUIVALENCE STATEMENTS

Use of EQUIVALENCE statements permits the sharing of the same storage unit by two or more entities. The general form of the statement is as follows:

```
EQUIVALENCE (u1),(u2),...,(un)
```

where each ui represents a sequence of two or more variables or array elements, separated by commas. Each element in the sequence is assigned the same storage unit (or portion of a storage unit) by the processor. The order in which the elements appear is not significant.

Example:

```
EQUIVALENCE (A,B,C)
```

The variables A, B and C will share the same storage unit during object program execution.

If an array element is used in an EQUIVALENCE statement, the number of subscripts must be the same as the number of dimensions established by the array declarator, or it must be one, where the one subscript specifies the array element's number relative to the first element of the array.

Example:

If the dimensionaliity of an array, Z, has been declared as Z(3,3) then in an EQUIVALENCE statement Z(6) and Z(3,2) have the same meaning.

Additonal Considerations:

1. The subscripts of array elements must be integer constants.

2. An element of a multi-dimensional array may be referred to by a single subscript, if desired.

3. Variables may be assigned to a COMMON block through EQUIVALENCE statements.

   Example:

```
COMMON /X/A,B,C
EQUIVALENCE (A,D)
```

   In this case, the variables A and D share the first storage unit in COMMON block X.

4. EQUIVALENCE statements can increase the size of a block indicated by a COMMON statement by adding more elements to the end of the block.

   Example:

```
DIMENSION R(2,2)
COMMON /Z/W,X,Y
EQUIVALENCE (Y,R(3))
```

   The resulting COMMON block will have the following configuration:

Variable	Storage Unit
W = R(1,1)	0
X = R(2,1)	1
Y = R(1,2)	2
R(2,2)	3

   The COMMON block established by the COMMON statement contains 3 storage units. It is expanded to 4 storage units by the EQUIVALENCE statement.

   COMMON block size may be increased only from the last element established by the COMMON statement forward; not from its first element backward.

Note that EQUIVALENCE (X,R(3)) would be invalid in the example. The COMMON statement established W as the first element in the COMMON block and an attempt to make X and R(3) equivalent would be an attempt to make R(1) the first element.

5. It is invalid to EQUIVALENCE two elements of the same array or two elements belonging to the same or different COMMON blocks.

   Example:

```
      DIMENSION XTABLE (20), D(5)
      COMMON A,B(4)/ZAP/C,X
            .
      .
      .
       EQUIVALENCE (XTABLE (6),A(7),
     X         B(3),XTABLE(15)),
     Y         (B(3),D(5))
      .
      .
      .
```

This EQUIVALENCE statement has the following errors:

1. It attempts to EQUIVALENCE two elements of the same array, XTABLE(6) and XTABLE(15).

2. It attempts to EQUIVALENCE two elements of the same COMMON block, A(7) and B(3).

3. Since A is not an array, A(7) is an illegal reference.

4. Making B(3) equivalent to D(5) extends COMMON backwards from its defined starting point.

6.8 DATA INITIALIZATION STATEMENT

The DATA initialization statement is a non-executable statement which provides a means of compiling data values into the object program and assigning these data to variables and array elements referenced by other statements.

The statement is of the following form:

DATA list/u1,u2,...,un/,list.../uk,uk+1,...uk+n/

where "list" represents a list of variable, array or array element names, and the ui are constants

corresponding in number to the elements in the list. An exception to the one-for-one correspondence of list items to constants is that an array name (unsubscripted) may appear in the list, and as many constants as necessary to fill the array may appear in the corresponding position between slashes. Instead of ui, it is permissible to write k*ui in order to declare the same constant, ui, k times in succession. k must be a positive integer. Dummy arguments may not appear in the list.

Example:

```
      DIMENSION C(7)
      DATA A, B, C(1),C(3)/14.73,
     X           -8.1,2*7.5/
```

This implies that

A=14.73, B=-8.1, C(1)=7.5, C(3)=7.5

The type of each constant ui must match the type of the corresponding item in the list, except that a Hollerith or Literal constant may be paired with an item of any type.

When a Hollerith or Literal constant is used, the number of characters in its string should be no greater than four times the number of storage units required by the corresponding item, i.e., 1 character for a Logical variable, up to 2 characters for an Integer variable and 4 or fewer characters for a Real variable.

If fewer Hollerith or Literal characters are specified, trailing blanks are added to fill the remainder of storage.

Hexadecimal data are stored in a similar fashion. If fewer Hexadecimal characters are used, sufficient leading zeros are added to fill the remainder of the storage unit.

The examples below illustrate many of the features of the DATA statement.

```
      DIMENSION HARY (2)
      DATA HARY,B/  4HTHIS, 4H OK.
     1          ,7.86/


      REAL  LIT(2)
      LOGICAL LT,LF
      DIMENSION  H4(2,2),PI3(3)
      DATA A1,B1,K1,LT,LF,H4(1,1),H4(2,1),
     1    H4(1,2),H4(2,2),PI3/5.9,2.5E-4,
     2      64,.FALSE.,.TRUE.,1.75E-3,
     3      0.85E-1,2*75.0,1.,2.,3.14159/,
     4      LIT(1)/'NOGO'/
```

6.9 IMPLICIT STATEMENT

The IMPLICIT statement is used to redefine default variable types. The syntax is:

```
IMPLICIT type(range),type(range),...
```

where type is one of the following: INTEGER, REAL, LOGICAL, DOUBLE PRECISION, BYTE, INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, REAL*8

and range is a list of alphabetic characters separated by commas or hyphens.

Examples:

```
IMPLICIT INTEGER(A,W-Z),REAL(B-V)
```

All variables (not otherwise declared) starting with the letters A, W, X, Y, Z will be type INTEGER. All variables starting with the letters B through V will be type REAL.

```
IMPLICIT INTEGER(I-N),REAL(A-H,O-Z)
```

This is the default definition.

Any IMPLICIT statements must appear in a program grouped with the Type and DIMENSION statements. IMPLICIT statements must appear before any other specification statements. If the IMPLICIT statement appears after any Type or DIMENSION statements, the types of the variables already declared will not be affected.

SECTION 7

FORTRAN CONTROL STATEMENTS

FORTRAN control statements are executable statements which affect and guide the logical flow of a FORTRAN program. The statements in this category are as follows:

1. GO TO statements:
    1. Unconditional GO TO
    2. Computed GO TO
    3. Assigned GO TO
2. ASSIGN
3. IF statements:
    1. Arithmetic IF
    2. Logical IF
4. DO
5. CONTINUE
6. STOP
7. PAUSE
8. CALL
9. RETURN
10. END

When statement labels of other statements are a part of a control statement, such statement labels must be associated with executable statements within the same program unit in which the control statement appears.

7.1 GO TO STATEMENTS

7.1.1 UNCONDITIONAL GO TO

Unconditional GO TO statements are used whenever control is to be transferred unconditionally to some other statement within the program unit.

The statement is of the following form:

```
GO TO k
```

where k is the statement label of an executable statement in the same program unit.

Example:

```
      GO TO 376
310   A(7) = V1 -A(3)
        .
        .
376   A(2) =VECT
      GO TO 310
```

In these statements, statement 376 is ahead of statement 310 in the logical flow of the program of which they are a part.

7.1.2 COMPUTED GO TO

Computed GO TO statements are of the form:

```
GO TO (k1,k2,...,n),j
```

where the ki are statement labels, and j is an integer variable, $1 \leq j \leq n$.

This statement causes transfer of control to the statement labeled kj. If $j \leq 1$ or $j > n$, control will be passed to the next statement following the Computed GOTO.

Example:

```
      J=3
      .
      .
      .
      GO TO(7, 70, 700, 7000, 70000), J
310   J=5
      GO TO 325
```

When J = 3, the computed GO TO transfers control to statement 700. Changing J to equal 5 changes the transfer to statement 70000. Making J = 0 or J = 6 would cause control to be transferred to statement 310.

7.1.3 ASSIGNED GO TO

Assigned GO TO statements are of the following form:

```
GO TO j,(k1,k2,...,kn)

  or

GOTO J
```

where J is an integer variable name, and the ki are statement labels of executable statements. This statement causes transfer of control to the statement whose label is equal to the current value of J.

Qualifications

1. The ASSIGN statement must logically precede an assigned GO TO.

2. The ASSIGN statement must assign a value to J which is a statement label included in the list of k's, if the list is specified.

Example:

```
GO TO LABEL,(80,90, 100)
```

Only the statement labels 80, 90 or 100 may be assigned to LABEL.

7.2 ASSIGN STATEMENT

This statement is of the following form:

```
ASSIGN j TO i
```

where j is a statement label of an executable statement and i is an integer variable.

The statement is used in conjunction with each assigned GO TO statement that contains the integer variable i. When the assigned GO TO is executed, control will be transferred to the statement labeled j.

Example:

```
ASSIGN 100 TO LABEL
.
.
.
ASSIGN 90 TO LABEL
GO TO LABEL, (80,90,100)
```

7.3 IF STATEMENT

IF statements transfer control to one of a series of statements depending upon a condition. Two types of IF statements are provided:

Arithmetic IF
Logical IF

7.3.1 ARITHMETIC IF

The arithmetic IF statement is of the form:

```
IF(e) m1,m2,m3
```

where e is an arithmetic expression and m1, m2 and m3 are statement labels.

Evaluation of expression e determines one of three transfer possibilities:

If e is:	Transfer to:
< 0	m1
= 0	m2
> 0	m3

Examples:

Statement	Expression Value	Transfer to
IF (A)3,4,5	15	5
IF (N-1)50,73,9	0	73
IF (AMTX(2,1,2))7,2,1	-256	7

7.3.2 LOGICAL IF

The Logical IF statement is of the form:

```
IF (u)s
```

where u is a Logical expression and s is any executable statement except a DO statement (see 7.4) or another Logical IF statement. The Logical expression u is evaluated as .TRUE. or .FALSE. Section 4 contains a discussion of Logical expressions.

Control Conditions:

If u is FALSE, the statement s is ignored and control goes to the next statement following the Logical IF statement. If, however, the expression is TRUE, then control goes to the statement s, and subsequent program control follows normal conditions.

If s is a replacement statement (v = e, Section 5), the variable and equality sign (=) must be on the same line, either immediately following IF(u) or on a separate continuation line with the line spaces following IF(u) left blank. See example 4 below.

Examples:

```
1.  IF(I.GT.20) GO TO 115

2.  IF(Q.AND.R) ASSIGN 10 TO J

3.  IF(Z) CALL DECL(A,B,C)

4.  IF(A.OR.B.LE.PI/2)I=J

5.  IF(A.OR.B.LE.PI/2)
   X    I =J
```

7.4 DO STATEMENT

The DO statement, as implemented in FORTRAN, provides a method for repetitively executing a series of statements. The statement takes of one of the two following forms:

1) `DO k i = m1,m2,m3`

or

2) `DO k i = m1,m2`

where k is a statement label, i is an integer or logical variable, and m1, m2 and m3 are integer constants or integer or logical variables.

If m3 is 1, it may be omitted as in 2) above.

The following conditions and restrictions govern the use of DO statements:

1. The DO and the first comma must appear on the initial line.

2. The statement labeled k, called the terminal statement, must be an executable statement.

3. The terminal statement must physically follow its associated DO, and the executable statements following the DO, up to and including the terminal statement, constitute the range of the DO statement.

4. The terminal statement may not be an Arithmetic IF, GO TO, RETURN, STOP, PAUSE or another DO.

5. If the terminal statement is a logical IF and its expression is .FALSE., then the statements in the DO range are reiterated.

   If the expression is .TRUE., the statement of the logical IF is executed and then the statements in the DO range are reiterated. The statement of the logical IF may not be a GO TO, Arithmetic IF, RETURN, STOP or PAUSE.

6. The controlling integer variable, i, is called the index of the DO range. The index must be positive and may not be modified by any statement in the range.

7. If m1, m2, and m3 are Integer*1 variables or constants, the DO loop will execute faster and be shorter, but the range is limited to 127

iterations. For example, the loop overhead for a DO loop with a constant limit and an increment of 1 depends upon the type of the index variable as follows:

Index Variable Type	Overhead Microseconds	Bytes
INTEGER*2	35.5	19
INTEGER*1	24	14

8. During the first execution of the statements in the DO range, i is equal to m1; the second execution, i = m1+m3; the third, i=m1+2*m3, etc., until i is equal to the highest value in this sequence less than or equal to m2, and then the DO is said to be satisfied. The statements in the DO range will always be executed at least once, even if m1 $\leq$ m2.

When the DO has been satisfied, control passes to the statement following the terminal statement, otherwise control transfers back to the first executable statement following the DO statement.

Example:

The following example computes

$$\sum_{i=1}^{100} A_i$$ where a is a one-dimensional array

```
100   DIMENSION A(100)
      .
      .
      .
      SUM = A(1)
      DO 31 I = 2,100
 31   SUM =SUM + A(I)

      END
```

9. The range of a DO statement may be extended to include all statements which may logically be executed between the DO and its terminal statement. Thus, parts of the DO range may be situated such that they are not physically between the DO statement and its terminal statement but are executed logically in the DO range. This is called the extended range.

Example:

```
      DIMENSION A(500), B(500)
.
.
.
      DO 50 I = 10, 327, 3
.
.
.
      IF (V7 -C*C) 20,15,31
30        .
.
.
50    A(I) = B(I) + C
.
.
.
20    C = C - .05
      GO TO 50
31    C=C+ .0125
      GO TO 30
```

10. It is invalid to transfer control into the range of a DO statement not itself in the range or extended range of the same DO statement.

11. Within the range of a DO statement, there may be other DO statements, in which case the DO's must be nested. That is, if the range of one DO contains another DO, then the range of the inner DO must be entirely included in the range of the outer DO.

    The terminal statement of the inner DO may also be the terminal statement of the outer DO.

For example, given a two dimensional array A of 15 rows and 15 columns, and a 15 element one-dimensional array B, the following statements compute the 15 elements of array C to the formula:

$$Ck = \sum_{j=1}^{15} AkjBm, \quad k = 1,2,\ldots,15$$

```
      DIMENSION A(15,15), B(15), C(15)
      .
      .
      .
      DO 80 K =1,15
      C(K) = 0.0
      DO 80 J=1,15
80    C(K) = C(K) +A(K,J) * B(J)
      .
      .
      .
```

7.5 CONTINUE STATEMENT

CONTINUE is classified as an executable statement. However, its execution does nothing. The form of the CONTINUE statement is as follows:

```
CONTINUE
```

CONTINUE is frequently used as the terminal statement in a DO statement range when the statement which would normally be the terminal statement is one of those which are not allowed or is only executed conditionally.

Example:

```
      DO 5 K = 1,10
      .
      .
      .
      IF (C2) 5,6,6
6     CONTINUE
      .
      .
      .
      C2 = C2 +.005
5     CONTINUE
```

7.6 STOP STATEMENT

A STOP statement has one of the following forms:

```
STOP

or

STOP c
```

where c is any string of one to six characters.

When STOP is encountered during execution of the object program, the characters c (if present) are displayed on the operator control console and execution of the program terminates.

The STOP statement, therefore, constitutes the logical end of the program.

7.7 PAUSE STATEMENT

A PAUSE statement has one of the following forms:

```
PAUSE

or

PAUSE c
```

where c is any string of up to six characters.

When PAUSE is encountered during execution of the object program, the characters c (if present) are displayed on the operator control console and execution of the program ceases.

The decision to continue execution of the program is not under control of the program. If execution is resumed through intervention of an operator without otherwise changing the state of the processor, the normal execution sequence, following PAUSE, is continued.

Execution may be terminated by typing a "T" at the operator console. Typing any other character will cause execution to resume.

7.8 CALL STATEMENT

CALL statements control transfers into SUBROUTINE subprograms and provide parameters for use by the subprograms. The general forms and detailed discussion of CALL statements appear in Section 9, FUNCTIONS AND SUBPROGRAMS.

7.9 RETURN STATEMENT

The form, use and interpretation of the RETURN statement is described in Section 9.

7.10 END STATEMENT

The END statement must physically be the last statement of any FORTRAN program. It has the following form:

```
END
```

The END statement is an executable statement and may have a statement label. It causes a transfer of control to be made to the system exit routine $EX, which returns control to the operating system.

SECTION 8

INPUT / OUTPUT

FORTRAN provides a series of statements which define the control and conditions of data transmission between computer memory and external data handling or mass storage devices such as magnetic tape, disk, line printer, punched card processors, keyboard printers, etc.

These statements are grouped as follows:

1. Formatted READ and WRITE statements which cause formatted information to be transmitted between the computer and I/O devices.

2. Unformatted READ and WRITE statements which transmit unformatted binary data in a form similar to internal storage.

3. Auxiliary I/O statements for positioning and demarcation of files.

4. ENCODE and DECODE statements for transferring data between memory locations.

5. FORMAT statements used in conjunction with formatted record transmission to provide data conversion and editing information between internal data representation and external character string forms.

8.1 FORMATTED READ/WRITE STATEMENTS

8.1.1 FORMATTED READ STATEMENTS

A formatted READ statement is used to transfer information from an input device to the computer.

Two forms of the statement are available, as follows:

```
READ (u,f,ERR=L1,END=L2) k
```

or

```
READ (u,f,ERR=L1,END=L2)
```

where:

u - specifies a Physical and Logical Unit Number and may be either an unsigned integer or an integer variable in the range 1 through 10. If an Integer variable is used, an Integer value must be assigned to it prior to execution of the READ statement.

LUN device assignments, along with their valid I/O operations, are listed below:

LUN	DEVICE	Valid I/O Operations
1	Console	Formatted read Formatted write Unformatted write
2	Line printer	Formatted write Unformatted write Endfile
3	Console	Same as LUN #1 above. Used by the runtime system for error messages, etc., so LUN #3 should usually be used for all console I/O.
4	PUN: and RDR: (Apple slot 2)	Formatted read Formatted write Unformatted write
5	Console	Same as LUN #1 above.
6-10	Disk file (FORT06.DAT-- FORT10.DAT)	Formatted read Formatted write Unformatted read Unformatted write Backspace Endfile

f - is the statement label of the FORMAT statement describing the type of data conversion to be used within the input transmission or it may be an array name, in which case the formatting information may be input to the program at the execution time. (See Section 8.7.10)

L1- is the FORTRAN label on the statement to which the I/O processor will transfer control if an I/O error is encountered.

L2- is the FORTRAN label on the statement to which the I/O processor will transfer control if an End-of-File is encountered.

k - is a list of variable names, separated by commas, specifying the input data.

READ (u,f)k is used to input a number of items, corresponding to the names in the list k, from the file on logical unit u, and using the FORMAT statement f to specify the external representation of these items (see FORMAT statements, 8.7). The ERR= and END= clauses are optional. If not specified, I/O errors and End-of-Files cause fatal runtime errors.

The following notes further define the function of the READ (u,f)k statement:

1. Each time execution of the READ statement begins, a new record from the input file is read.

2. The number of records to be input by a single READ statement is determined by the list, k, and format specifications.

3. The list k specifies the number of items to be read from the input file and the locations into which they are to be stored.

4. Any number of items may appear in a single list and the items may be of different data types.

5. If there are more quantities in an input record than there are items in the list, only the number of quantities equal to the number of items in the list are transmitted. Remaining quantities are ignored.

6. Exact specifications for the list k are described in 8.6.

Examples:

1. Assume that four Integer data entries are stored in a sequential disk file and that the values have field widths of 3, 4, 2 and 5 respectively. The statements

```
      READ(5,20)K,L,M,N
   20 FORMAT(I3,I4,I2,I5)
```

   will read the file and assign the input data to the variables K, L, M and N.

   See 8.7 for complete description of FORMAT statements.

2. Input the quantities of an array (ARRY):

```
      READ(6,21)ARRY
```

   Only the name of the array needs to appear in the list (see 8.6). All elements of the array ARRY will be read and stored using the appropriate formatting specified by the FORMAT statement labeled 21.

READ(u,k) may be used in conjunction with a FORMAT statement to read H-type alphanumeric data into an existing H-type field (see Hollerith Conversions, 8.7.3).

For example, the statements

```
      READ(I,25)
      .
      .
      .
   25 FORMAT(10HABCDEFGHIJ)
```

cause the next 10 characters of the file on input device I to be read and replace the characters ABCDEFGHIJ in the FORMAT statement.

8.1.2 FORMATTED WRITE STATEMENTS

A formatted WRITE statement is used to transfer information from the computer to an output device.

Two forms of the statement are available, as follows:

```
WRITE(u,f,ERR=L1,END=L2)k

or

WRITE (u,f,ERR=L1,END=L2)
```

where:

u - specifies a Logical Unit Number.

f - is the statement label of the FORMAT statement describing the type of data conversion to be used with the output transmission.

L1- specifies an I/O error branch.

L2- specifies an EOF branch.

k - is a list of variable names separated by commas, specifying the output data.

WRITE (u,f)k is used to output the data specified in the list k to a file on logical unit u using the FORMAT statement f to specify the external representation of the data (see FORMAT statements, 8.7). The following notes further define the function of the WRITE statement:

1. Several records may be output with a single WRITE statement, with the number determined by the list and FORMAT specifications.

2. Successive data are output until the data specified in the list are exhausted.

3. If output is to a device which specifies fixed length records and the data specified in the list do not fill the record, the remainder of the record is filled with blanks.

Example:

```
WRITE(2,10)A,B,C,D
```

The data assigned to the variables A, B, C and D are output to Logical Unit Number 2, formatted according to the FORMAT statement labeled 10.

WRITE(u,f) may be used to write alphanumeric information when the characters to be written are specified within the FORMAT statement. In this case a variable list is not required.

For example, to write the characters 'H CONVERSION' on unit 1,

```
      WRITE(1,26)
      .
      .
      .
   26 FORMAT ('H CONVERSION')
```

8.2 UNFORMATTED READ/WRITE

Unformatted I/O (i.e. without data conversion) is accomplished using the statements:

READ(u,ERR=L1,END=L2) k

WRITE(u,ERR=L1,END=L2) k

where:

u - specifies a Logical Unit Number.

L1- specifies an I/O error branch.

L2- specifies an EOF branch.

k - is a list of variable names, separated by commas, specifying the I/O data.

The following notes define the functions of unformatted I/O statements.

1. Unformatted READ/WRITE statements perform memory-image transmission of data with no data conversion or editing.

2. The amount of data transmitted corresponds to the number of variables in the list k.

3. The total length of the list of variable names in an unformatted READ must not be longer than the record length. If the logical record length and the length of the list are the same, the entire record is read. If the length of the list is shorter than the logical record length the unread items in the record are skipped.

4. The WRITE(a)k statement writes one logical record.

5. A logical record may extend across more than one physical record.

6. When performing an Unformatted WRITE to a serial device (such as a printer or console), there is no limitation on the amount of data sent. For example, an entire array could be output to the screen with an unformatted WRITE statement:

```
BYTE CHR(80,24)
  .
  .
  .
WRITE (3) CHR
```

8.3 DISK FILE I/O

A READ or WRITE to a disk file (LUN 6-10) automatically OPENs the file for I/O. The file remains open until closed by an ENDFILE command (see Section 8.4) or until normal program termination.

NOTE

Exercise caution when doing sequential output to disk files. If output is done to an existing file, the existing file will be deleted and replaced with a new file of the same name.

8.3.1 RANDOM DISK I/O

SEE ALSO CHAPTER 5 OF YOUR FORTRAN-80 USER'S MANUAL.

FORTRAN-80 supports random access I/O to disk files with one limitation: the record length is fixed at 128 bytes. If a random access WRITE does not transfer enough data to fill the record to 128 bytes, the rest of the record is filled with zeros (NUL characters). Care must be taken to insure that the data written do not exceed 128 bytes.

For random disk access, the record number is specified by using the REC=n option in the READ or WRITE statement. For example:

```
I = 10
WRITE (6,20,REC=I,ERR=50) X, Y, Z
   .
   .
   .
```

This program segment writes record 10 on LUN 6. If a previous record 10 exists, it is written over. If no record 10 exists, the file is extended to create one. Any attempt to read a non-existent record results in an I/O error.

In order to insure the integrity of random disk files, it is a good idea to always use ENDFILE or REWIND after WRITEing data.

A disk file that is OPENed by a READ or WRITE statement is given a default name that depends upon the LUN and the operating system. For CP/M, the default filenames for LUNs 6 through 10 are:

FORT06.DAT, FORT07.DAT,..., FORT10.DAT

To explicitly name a disk file, open it by calling the OPEN subroutine.

8.3.2 OPEN SUBROUTINE

Alternatively, a file may be OPENed using the OPEN subroutine. LUNs 1-5 may also be assigned to disk files with OPEN. The OPEN subroutine allows the program to specify a filename and device to be associated with a LUN.

An OPEN of a non-existent file creates a null file of the appropriate name. An OPEN of an existing file followed by sequential output deletes the existing file. An OPEN of an existing file followed by an input allows access to the current contents of the file.

The format of an OPEN call under CP/M is:

```
CALL OPEN (LUN, Filename, Drive)
```

where:

LUN = a Logical Unit Number to be associated with the file (must be an Integer value between 1 and 10).

Filename = an ASCII name which the operating system will associate with the file. The Filename should be a Hollerith or Literal constant, or a variable or array name, where the variable or array contains the ASCII name. The Filename should be blank-filled to exactly eleven characters.

Drive = the disk drive on which the file exists or will exist (must be an Integer constant or Integer variable within the range allowed by the operating system).

Example:

```
CALL OPEN (6,'PROG2   FOR',0)
```

Note that "0" indicates the currently logged drive, "1" indicates drive A:, and so on. Note that the filename and extension use exactly 11 spaces. The filename is left-justified in the first 8 spaces, the extension is right-justified in the last 3 spaces, and blanks are inserted to make 11 characters.

8.4 AUXILIARY I/O STATEMENTS

Three auxiliary I/O statements are provided:

```
BACKSPACE u
REWIND u
ENDFILE u
```

The actions of all three statements depend on the LUN with which they are used (see Appendix B). When the LUN is for a terminal or line printer, the three statements are defined as no-ops.

When the LUN is for a disk drive, the ENDFILE and REWIND commands allow further program control of disk files. ENDFILE u closes the file associated with LUN u. REWIND u closes the file associated with LUN u, then opens it again. BACKSPACE is not implemented at this time, and therefore causes an error if used.

8.5 ENCODE/DECODE

ENCODE and DECODE statements transfer data, according to format specifications, from one section of memory to another. DECODE changes data from ASCII format to the specified format. ENCODE changes data of the specified format into ASCII format. The two statements are of the form:

```
ENCODE(a,f) k
DECODE(a,f) k
```

where:

```
a is an array name
f is FORMAT statement number
k is an I/O List
```

DECODE is analogous to a READ statement, since it causes conversion from ASCII to internal format. ENCODE is analogous to a WRITE statement, causing conversion from internal formats to ASCII.

NOTE

Care should be taken that the array A is always large enough to contain all of the data being processed. There is no check for overflow. An ENCODE operation which overflows the array will probably wipe out important data following the array. A DECODE operation which overflows will attempt to process the data following the array.

8.6 INPUT/OUTPUT LIST SPECIFICATIONS

Most forms of READ/WRITE statements may contain an ordered list of data names which identify the data to be transmitted. The order in which the list items appear must be the same as that in which the corresponding data exists (Input), or will exist (Output) in the external I/O medium.

Lists have the following form:

```
m1,m2,...,mn
```

where the mi are list items separated by commas, as shown.

8.6.1 LIST ITEM TYPES

A list item may be a single datum identifier or a multiple data identifier.

1. A single datum identifier item is the name of a variable or array element.

   Examples:

```
A
C(26,1),R,K,D
B,I(10,10),S,F(1,25)
```

NOTE

Sublists are not implemented.

2. Multiple data identifier items are in two forms:

   a. An array name appearing in a list without subscript(s) is considered equivalent to the listing of each successive element of the array.

   Example:

   If B is a two dimensional array, the list item B is equivalent to: B(1,1),B(2,1),B(3,1)...., B(1,2),B(2,2)...,B(j,k).

   where j and k are the subscript limits of B.

   b. DO-implied items are lists of one or more single datum identifiers or other DO-implied items followed by a comma character and an expression of the form:

   i = m1,m2,m3 or i = m1,m2

   and enclosed in parentheses.

   The elements i,m1,m2,m3 have the same meaning as defined for the DO statement. The DO implication applies to all list items enclosed in parentheses with the implication.

   Examples:

DO-Implied Lists	Equivalent Lists
(X(I),I=1,4)	X(1),X(2),X(3),X(4)
(Q(J),R(J),J=1,2)	Q(1),R(1),Q(2),R(2)
(G(K),K=1,7,3)	G(1),G(4),G(7)
((A(I,J),I=3,5),J=1,9,4)	A(3,1),A(4,1),A(5,1) A(3,5),A(4,5),A(5,5) A(3,9),A(4,9),A(5,9)
(R(M),M=1,2),I,ZAP(3)	R(1),R(2),I,ZAP(3)
(R(3),T(I),I=1,3)	R(3),T(1),R(3),T(2), R(3),T(3)

Thus, the elements of a matrix, for example, may be transmitted in an order different from the order in which they appear in storage. The array A(3,3) occupies storage in the order A(1,1),A(2,1), A(3,1),A(1,2),A(2,2),A(3,2), A(1,3),A(2,3),A(3,3). By specifying the transmission of the array with the DO-implied list item ((A(I,J),J=1,3),I=1,3), the order of transmission is:
A(1,1),A(1,2),A(1,3),A(2,1),A(2,2),
A(2,3),A(3,1),A(3,2),A(3,3)

8.6.2 SPECIAL NOTES ON LIST SPECIFICATIONS

1. The ordering of a list is from left to right with repetition of items enclosed in parentheses (other than as subscripts) when accompanied by controlling DO-implied index parameters.

2. Arrays are transmitted by the appearance of the array name (unsubscripted) in an input/output list.

3. Constants may appear in an input/output list only as subscripts or as indexing parameters.

4. For input lists, the DO-implying elements i, m1, m2 and m3 may not appear within the parentheses as list items.

Examples:

1. READ (1,20) (I,J,A(I),I=1,J,2) is not allowed

2. READ(1,20)I,J,(A(I),I=1,J,2) is allowed

3. WRITE(1,20)(I,J,A(I),I=1,J,2) is allowed

Consider the following examples:

```
DIMENSION A(25)

A(1) = 2.1
A(3) = 2.2
A(5) = 2.3
J = 5

WRITE (1,20) J,(I,A(I),I=1,J,2)
.
.
.
```

the output of this WRITE statement is

```
5,1,2.1,3,2.2,5,2.3
```

1. Any number of items may appear in a single list.

2. In a formatted transmission (READ(u,f)k, WRITE(u,f)k) each item must have the correct type as specified by a FORMAT statement.

8.7 FORMAT STATEMENTS

FORMAT statements are non-executable, generative statements used in conjunction with formatted READ and WRITE statements. They specify conversion methods and data editing information as the data is transmitted between computer storage and external media representation.

FORMAT statements require statement labels for reference (f) in the READ(u,f)k or WRITE(u,f)k statements.

The general form of a FORMAT statement is as follows:

```
m FORMAT (s1,s2,...,sn/s1',s2',...,sn'/...)
```

where m is the statement label and each si is a field descriptor. The word FORMAT and the parentheses must be present as shown. The slash (/) and comma (,) characters are field separators and are described in a separate subparagraph. The field is defined as that part of an external record occupied by one transmitted item.

8.7.1 FIELD DESCRIPTORS

Field descriptors describe the sizes of data fields and specify the type of conversion to be exercised upon each transmitted datum. The FORMAT field descriptors may have any of the following forms:

Descriptor	Classification
rFw.d rGw.d rEw.d rDw.d rIw	Numeric Conversion
rLw	Logical Conversion
rAw nHh1h2...hn 'l1l2...ln'	Hollerith Conversion
nX	Spacing Specification
mP	Scaling Factor

where:

1. w and n are positive integer constants defining the field width (including digits, decimal points, algebraic signs) in the external data representation.

2. d is an integer specifying the number of fractional digits appearing in the external data representation.

3. The characters F, G, E, D, I, A and L indicate the type of conversion to be applied to the items in an input/output list.

4. r is an optional, non-zero integer indicating that the descriptor will be repeated r times.

5. The hi and li are characters from the FORTRAN character set.

6. m is an integer constant (positive, negative, or zero) indicating scaling.

8.7.2 NUMERIC CONVERSIONS

Input operations with any of the numeric conversions will allow the data to be represented in a "Free Format"; i.e., commas may be used to separate the fields in the external representation.

F-type conversion

Form: Fw.d

Real or Double Precision type data are processed using this conversion. w characters are processed of which d are considered fractional.

F-output

Values are converted and output as minus sign (if negative), followed by the integer portion of the number, a decimal point and d digits of the fractional portion of the number. If a value does not fill the field, it is right justified in the field and enough preceding blanks to fill the field are inserted. If a value requires more field positions than allowed by w, the first w-1 digits of the value are output, preceded by an asterisk.

F-Output Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
F10.4	368.42	bb368.4200
F7.1	-4786.361	-4786.4
F8.4	8.7E-2	bb0.0870
F6.4	4739.76	*.7600
F7.3	-5.6	b-5.600

* Note the loss of leading digits in the 4th line above.

F-Input

(See the description under E-Input below.)

E-type Conversion

Form: Ew.d

Real or Double Precision type data are processed using this conversion. w characters are processed of which d are considered fractional.

E-Output

Values are converted, rounded to d digits, and output as:

1. a minus sign (if negative),

2. a zero and a decimal point,

3. d decimal digits,

4. the letter E,

5. the sign of the exponent (minus or blank),

6. two exponent digits,

in that order. The values as described are right justified in the field w with preceding blanks to fill the field if necessary. The field width w should satisfy the relationship:

w > d + 7

Otherwise significant characters may be lost. Some E-Output examples follow:

FORMAT Descriptor	Internal Value	Output (b=blank)
E12.5	76.573	bb.76573Eb02
E14.7	-32672.354	-b.3267235Eb05
E13.4	-0.0012321	bb-b.1232E-02
E8.2	76321.73	b.76Eb05

E-Input

Data values which are to be processed under E, F, or G conversion can be a relatively loose format in the external input medium. The format is identical for either conversion and is as follows:

1. Leading spaces (ignored)

2. A + or - sign (an unsigned input is assumed to be positive)

3. A string of digits

4. A decimal point

5. A second string of digits

6. The character E

7. A + or - sign

8. A decimal exponent

Each item in the list above is optional; but the following conditions must be observed:

1. If FORMAT items 3 and 5 (above) are present, then 4 is required.

2. If FORMAT item 8 is present, then 6 or 7 or both are required.

3. All non-leading spaces are considered zeros.

Input data can be any number of digits in length, and correct magnitudes will be developed, but precision will be maintained only to the extent specified in Section 3 for Real data.

E- and F- and G- Input Examples:

FORMAT Descriptor	Input (b=blank)	Internal Value
E10.3	+0.23756+4	+2375.60
E10.3	bbbbb17631	+17.631
G8.3	b1628911	+1628.911
F12.4	bbbb-6321132	-632.1132

Note in the above examples that if no decimal point is given among the input characters, the d in the FORMAT specification establishes the decimal point in conjunction with an exponent, if given. If a decimal point is included in the input characters, the d specification is ignored.

The letters E, F, and G are interchangeable in the input format specifications. The end result is the same.

D-Type Conversions

D-Input and D-Output are identical to E-Input and E-Output except the exponent may be specified with a "D" instead of an "E."

G-Type Conversions

Form: Gw.d

Real or Double Precision type data are processed using this conversion. w characters are processed of which d are considered significant.

G-Input:

(See the description under E-Input)

G-Output:

The method of output conversion is a function of the magnitude of the number being output. Let n be the magnitude of the number. The following table shows how the number will be output:

Magnitude	Equivalent Conversion
.1 <= n < 1	F(w-4).d,4X
1 <= n < 10	F(w-4).(d-1),4X
.	.
.	.
.	.
10^{d-2} <= n < 10^{d-1}	F(w-4).1,4X
10^{d-1} <= n < 10^{d}	F(w-4).0,4X
Otherwise	Ew.d

I-Conversions

Form: Iw

Only Integer data may be converted by this form of conversion. w specifies field width.

I-Output:

Values are converted to Integer constants. Negative values are preceded by a minus sign. If the value does not fill the field, it is right justified in the field and enough preceding blanks to fill the field are inserted. If the value exceeds the field width, only the least significant w-1 characters are output preceded by an asterisk.

Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
I6	+281	bbb281
I6	-23261	-23261
I3	126	126
I4	-226	-226

I-Input:

A field of w characters is input and converted to internal integer format. A minus sign may precede the integer digits. If a sign is not present, the value is considered positive.

Integer values in the range -32768 to 32767 are accepted. Non-leading spaces are treated as zeros.

Examples:

Format Descriptor	Input (b=blank)	Internal Value
I4	b124	124
I4	-124	-124
I7	bb6732b	67320
I4	1b2b	1020

8.7.3 HOLLERITH CONVERSIONS

A-Type Conversion

The form of the A conversion is as follows:

Aw

This descriptor causes unmodified Hollerith characters to be read into or written from a specified list item.

The maximum number of actual characters which may be transmitted between internal and external representations using Aw is four times the number of storage units in the corresponding list item (i.e., 1 character for logical items, 2 characters for Integer items, 4 characters for Real items and 8 characters for Double Precision items).

A-Output:

If w is greater than 4n (where n is the number of storage units required by the list item), the

external output field will consist of w-4n blanks followed by the 4n characters from the internal representation. If w is less than 4n, the external output field will consist of the leftmost w characters from the internal representation.

Examples:

Format Descriptor	Internal	Type	Output (b=blanks)
A1	A1	Integer	A
A2	AB	Integer	AB
A3	ABCD	Real	ABC
A4	ABCD	Real	ABCD
A7	ABCD	Real	bbbABCD

A-Input:

If w is greater than 4n (where n is the number of storage units required by the corresponding list item), the rightmost 4n characters are taken from the external input field. If w is less than 4n, the w characters appear left justified with w-4n trailing blanks in the internal representation.

Examples:

Format Descriptor	Input Characters	Type	Internal (b=blank)
A1	A	Integer	Ab
A3	ABC	Integer	AB
A4	ABCD	Integer	AB
A1	A	Real	Abbb
A7	ABCDEFG	Real	DEFG

H-Conversion

The forms of H conversion are as follows:

nHh1h2...hn

'h1h2...hn'

These descriptors process Hollerith character strings between the descriptor and the external field, where each hi represents any character from the ASCII character set.

NOTE

Special consideration is required if an apostrophe (') is to be used within the literal string in the second form. An apostrophe character within the string is represented by two successive apostrophes. See the examples below.

H-Output:

The n characters hi, are placed in the external field. In the nHh1h2...hn form the number of characters in the string must be exactly as specified by n. Otherwise, characters from other descriptors will be taken as part of the string. In both forms, blanks are counted as characters.

Examples:

Format Descriptor			Output (b=blank)
1HA	or	'A'	A
8HbSTRINGb	or	'bSTRINGb'	bSTRINGb
11HX(2,3)=12.0	or	'X(2,3)=12.0'	X(2,3)=12.0
11HIbSHOULDN'T	or	'IbSHOULDN''T'	IbSHOULDN'T

H-Input

The n characters of the string hi are replaced by the next n characters from the input record. This results in a new string of characters in the field descriptor.

FORMAT Descriptor			Input (b=blank)	Resultant Descriptor		
4H1234	or	'1234'	ABCD	4HABCD	or	'ABCD'
7HbbFALSE	or	'bbFALSE'	bFALSEb	7HbFALSEb	or	'bFALSEb'
6Hbbbbbb	or	'bbbbbb'	MATRIX	6HMATRIX	or	'MATRIX'

8.7.4 LOGICAL CONVERSIONS

The form of the logical conversion is as follows:

Lw

L-Output:

If the value of an item in an output list

corresponding to this descriptor is 0, an F will be output; otherwise, a T will be output. If w is greater than 1, w-1 leading blanks precede the letters.

Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
L1	=0	F
L1	<>0	T
L5	<>0	bbbbT
L7	=0	bbbbbbF

L-Input

The external representation occupies w positions. It consists of optional blanks followed by a "T" or "F", followed by optional characters.

8.7.5 X DESCRIPTOR

The form of X conversion is as follows:

nX

This descriptor causes no conversion to occur, nor does it correspond to an item in an input/output list. When used for output, it causes n blanks to be inserted in the output record. Under input circumstances, this descriptor causes the next n characters of the input record to be skipped. Note that 1X is required, as X alone will not work.

Output Examples:

FORMAT Statement	Output (b=blank)
3 FORMAT (1HA,4X,2HBC)	AbbbbBC
7 FORMAT (3X,4HABCD,1X)	bbbABCDb

Input Examples:

FORMAT Statement	Input String	Resultant Input
10 FORMAT (F4.1,3X,F3.0)	12.5ABC120	12.5,120
5 FORMAT (7X,I3)	1234567012	012

8.7.6 P DESCRIPTOR

The P descriptor is used to specify a scaling factor for real conversions (F, E, D, G). The form is nP where n is an integer constant (positive, negative, or zero).

The scaling factor is automatically set to zero at the beginning of each formatted I/O call (each READ or WRITE statement). If a P descriptor is encountered while scanning a FORMAT, the scale factor is changed to n. The scale factor remains changed until another P descriptor is encountered or the I/O terminates.

Effects of Scale Factor on Input:

During E, F, or G input the scale factor takes effect only if no exponent is present in the external representation. In that case, the internal value will be a factor of 10**n less than the external value (the number will be divided by 10**n before being stored).

Effect of Scale Factor on Output:

E-Output, D-Output:

The coefficient is shifted left n places relative to the decimal point, and the exponent is reduced by n (the value remains the same).

F-Output:

The external value will be 10**n times the internal value.

G-Output:

The scale factor is ignored if the internal value is small enough to be output using F conversion. Otherwise, the effect is the same as for E output.

8.7.7 SPECIAL CONTROL FEATURES OF FORMAT STATEMENTS

8.7.7.1 Repeat Specifications

1. The E, F, D, G, I, L and A field descriptors may be indicated as repetitive descriptors by using a repeat count r in the form rEw.d, rFw.d, rGw.d, rIw, rLw, rAw. The following pairs of FORMAT statements are equivalent:

```
  66   FORMAT (3F8.3,F9.2)
C  IS EQUIVALENT TO:
   66  FORMAT (F8.3,F8.3,F8.3,F9.2)

   14  FORMAT (2I3,2A5,2E10.5)
C   IS EQUIVALENT TO:
   14  FORMAT (I3,I3,A5,A5,E10.5,E10.5)
```

2. Repetition of a group of field descriptors is accomplished by enclosing the group in parentheses preceded by a repeat count. Absence of a repeat count indicates a count of one. Up to two levels of parentheses, including the parentheses required by the FORMAT statement, are permitted.

   Note the following equivalent statements:

```
   22  FORMAT (I3,4(F6.1,2X))
C  IS EQUIVALENT TO:
   22  FORMAT (I3,F6.1,2X,F6.1,2X,F6.1,2X,
     1    F6.1,2X)
```

3. Repetition of FORMAT descriptors is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list that have not been processed. When this occurs the FORMAT descriptors are re-used starting at the first opening parenthesis in the FORMAT statement. A repeat count preceding the parenthesized descriptor(s) to be re-used is also active in the re-use. This type of repetitive use of FORMAT descriptors terminates processing of the current record and initiates the processing of a new record each time the re-use begins. Record demarcation under these circumstances is the same as in the paragraph 8.7.7.2 below.

Input Example:

```
      DIMENSION A(100)
      READ (3,13) A
      .
      .
      .
   13 FORMAT (5F7.3)
```

In this example, the first 5 quantities from each of 20 records are input and assigned to the array elements of the array A.

Output Example:

```
      .
      .
      .
      WRITE (6,12)E,F,K,L,M,KK,LL,MM,K3,L3,M3
      .
      .
      .
   12 FORMAT (2F9.4,3(3I7,/))
```

In this example, three records are written. Record 1 contains E, F, K, L and M. Because the descriptor 3I7 is reused twice, Record 2 contains KK, LL and MM and Record 3 contains K3, L3 and M3.

8.7.7.2 Field Separators

Two adjacent descriptors must be separated in the FORMAT statement by either a comma or one or more slashes.

Example:

2H0K/F6.3 or 2H0K,F6.3

The slash not only separates field descriptors, but it also specifies the demarcation of formatted records.

Each slash terminates a record and sets up the next record for processing. The remainder of an input record is ignored; the remainder of an output record is filled with blanks. Successive slashes (///.../) cause successive records to be ignored on input and successive blank records to be written on output.

Output example:

```
      DIMENSION A(100),J(20)
      .
      .
      .
      WRITE (7,8) J,A
  8   FORMAT (10I7/10I7/50F7.3/50F7.3)
```

In this example, the data specified by the list of the WRITE statement are output to unit 7 according to the specifications of FORMAT statement 8. Four records are written as follows:

Record 1	Record 2	Record 3	Record 4
J(1)	J(11)	A(1)	A(51)
J(2)	J(12)	A(2)	A(52)
.	.	.	.
.	.	.	.
.	.	.	.
J(10)	J(20)	A(50)	A(100)

Input Example:

```
      DIMENSION B(10)
      .
      .
      .
      READ (4,17) B
  17  FORMAT(F10.2/F10.2///8F10.2)
```

In this example, the two array elements B(1) and B(2) receive their values from the first data fields of successive records (the remainders of the two records are ignored). The third and fourth records are ignored and the remaining elements of the array are filled from the fifth record.

8.7.8 FORMAT CONTROL, LIST SPECIFICATIONS AND RECORD DEMARCATION

The following relationships and interactions between FORMAT control, input/output lists and record demarcation should be noted:

1. Execution of a formatted READ or WRITE statement initiates FORMAT control.

2. The conversion performed on data depends on information jointly provided by the elements in the input/output list and field descriptors in the FORMAT statement.

3. If there is an input/output list, at least one descriptor of types E, F, D, G, I, L or A must be present in the FORMAT statement.

4. Each execution of a formatted READ statement causes a new record to be input.

5. Each item in an input list corresponds to a string of characters in the record and to a descriptor of the types E, F, G, I, L or A in the FORMAT statement. .

6. H and X descriptors communicate information directly between the external record and the field descriptors without reference to list items.

7. On input, whenever a slash is encountered in the FORMAT statement or the FORMAT descriptors have been exhausted and re-use of descriptors is initiated, processing of the current record is terminated and the following occurs:

   a. Any unprocessed characters in the record are ignored.

   b. If more input is necessary to satisfy list requirements, the next record is read.

8. A READ statement is terminated when all items in the input list have been satisfied if:

   a. The next FORMAT descriptor is E, F, G, I, L or A.

   b. The FORMAT control has reached the last outer right parenthesis of the FORMAT statement.

   If the input list has been satisfied, but the next FORMAT descriptor is H or X, more data are processed (with the possibility of new records being input) until one of the above conditions exists.

9. If FORMAT control reaches the last right parenthesis of the FORMAT statement but there are more list items to be processed, all or part of the descriptors are reused. (See item 3 in the description of Repeat Specifications, sub-paragraph 8.7.7.1)

10. When a Formatted WRITE statement is executed, records are written each time a slash is encountered in the FORMAT statement or FORMAT control has reached the rightmost right parenthesis. The FORMAT control terminates in one of the two methods described for READ termination in 8 above. Incomplete records are filled with blanks to maintain record lengths.

8.7.9 FORMAT CARRIAGE CONTROL

Formatted I/O to a console or printer uses the first character of each record for carriage control. The carriage control character is never printed. The carriage control character determines what action will be taken before the line is printed. The options are as follows:

Control Character	Action Taken Before Printing
0	Skip 2 lines
1	Insert Form Feed
+	No advance
*	No action (The next WRITE will pick up where the last WRITE left off.)
Other	Skip 1 line

Formatted I/O to disk does not require the first character of each record to be a carriage control character. Records are terminated by a carriage return character (X'0D'). There are no line-feeds (X'0A') in the file unless written there explicitly.

8.7.10 FORMAT SPECIFICATIONS IN ARRAYS

The FORMAT reference, f, of a formatted READ or WRITE statement (See 8.1) may be an array name instead of a statement label. If such reference is made, at the time of execution of the READ/WRITE statement the first part of the information contained in the array, taken in natural order, must constitute a valid FORMAT specification. The array may contain non-FORMAT information following the right parenthesis that ends the FORMAT specification.

The FORMAT specification which is to be inserted in the array has the same form as defined for a FORMAT statement (i.e., it begins with a left parenthesis and ends with a right parenthesis).

The FORMAT specification may be inserted in the array by use of a DATA initialization statement, or by use of a READ statement together with an Aw FORMAT. Example:

Assume the FORMAT specification

```
(3F10.3,4I6)
```

or a similar 12 character specification is to be stored into an array. The array must allow a minimum of 3 storage units.

The FORTRAN coding below shows the various methods of establishing the FORMAT specification and then referencing the array for a formatted READ or WRITE.

```
C  DECLARE A REAL ARRAY
       DIMENSION A(3), B(3), M(4)
C  INITIALIZE FORMAT WITH DATA STATEMENT
       DATA A/'(3F1','0.3,',','4I6)'/
        .
        .
        .
C  READ DATA USING FORMAT SPECIFICATIONS
C       IN ARRAY A
       READ(6,A) B, M

C  DECLARE AN INTEGER ARRAY
       DIMENSION IA(4), B(3), M(4)
        .
        .
        .
C  READ FORMAT SPECIFICATIONS
       READ (7,15) IA
C  FORMAT FOR INPUT OF FORMAT SPECIFICATIONS
   15  FORMAT (4A2)
        .
        .
        .
C  READ DATA USING PREVIOUSLY INPUT
C       FORMAT SPECIFICATION
       READ (7,IA) B,M
        .
        .
        .
```

SECTION 9

FUNCTIONS AND SUBPROGRAMS

The FORTRAN language provides a means for defining and using often needed programming procedures such that the statement or statements of the procedures need appear in a program only once but may be referenced and brought into the logical execution sequence of the program whenever and as often as needed.

These procedures are as follows:

1. Statement functions.
2. Library functions.
3. FUNCTION subprograms.
4. SUBROUTINE subprograms.

Each of these procedures has its own unique requirements for reference and defining purposes. These requirements are discussed in subsequent paragraphs of this section. However, certain features are common to the whole group or to two or more of the procedures. These common features are as follows:

1. Each of these procedures is referenced by its name which, in all cases, is one to six alphanumeric characters of which the first is a letter.
2. The first three are designated as "functions" and are alike in that:
    1. They are always single valued (i.e., they return one value to the program unit from which they are referenced).
    2. They are referred to by an expression containing a function name.
    3. They must be typed by type specification statements if the data type of the single-valued result is to be different from that indicated by the pre-defined convention.
3. FUNCTION subprograms and SUBROUTINE subprograms are considered program units.

In the following descriptions of these procedures, the term calling program means the program unit or procedure in which a reference to a procedure is made, and the term "called program" means the procedure to which a reference is made.

9.1 THE PROGRAM STATEMENT

The PROGRAM statement provides a means of specifying a name for a main program unit. The form of the statement is:

```
PROGRAM name
```

If present, the PROGRAM statement must appear before any other statement in the program unit. The name consists of 1-6 alphanumeric characters, the first of which is a letter. If no PROGRAM statement is present in a main program, the compiler assigns a name of $MAIN to that program.

9.2 STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic or logical assignment statement and are relevant only to the program unit in which they appear. The general form of a statement function is as follows:

```
f(a1,a2,...an) = e
```

where f is the function name, the ai are dummy arguments and e is an arithmetic or logical expression.

Rules for ordering, structure and use of statement functions are as follows:

1. Statement function definitions, if they exist in a program unit, must precede all executable statements in the unit and follow all specification statements.

2. The ai are distinct variable names or array elements, but, being dummy variables, they may have the same names as variables of the same type appearing elsewhere in the program unit.

3. The expression e is constructed according to the rules in SECTION 4 and may contain only references to the dummy arguments and non-Literal constants, variable and array element references, utility and mathematical function references and references to previously defined statement functions.

4. The type of any statement function name or argument that differs from its pre-defined convention type must be defined by a type specification statement.

5. The relationship between f and e must conform to the replacement rules in Section 5.

6. A statement function is called by its name followed by a parenthesized list of arguments. The expression is evaluated using the arguments specified in the call, and the reference is replaced by the result.

7. The ith parameter in every argument list MUST agree in type with the ith dummy in the statement function.

The example below shows a statement function and a statement function call.

```
C  STATEMENT FUNCTION DEFINITION
C
      FUNC1(A,B,C,D) = ((A+B)**C)/D


C  STATEMENT FUNCTION CALL
C
      A12=A1-FUNC1(X,Y,Z7,C7)
```

9.3 LIBRARY FUNCTIONS

Library functions are a group of utility and mathematical functions which are "built-in" to the FORTRAN system. Their names are pre-defined to the Processor and automatically typed. The functions are listed in Tables 9-1 and 9-2. In the tables, arguments are denoted as a1,a2,...,an, if more than one argument is required; or as a if only one is required. A library function is called when its name is used in an arithmetic expression.

Such a reference takes the following form:

f(a1,a2,...an)

where f is the name of the function and the ai are actual arguments. The arguments must agree in type, number and order with the specifications indicated in Tables 9-1 and 9-2.
In addition to the functions listed in 9-1 and 9-2, two additional library subprograms enable direct access to the 8080 (or Z80) hardware. These are:

PEEK and POKE

For the following:
b, b1, and b2 are BYTE constants or variables
i is an INTEGER constant or variable

PEEK is a Logical function; POKE is a subroutine. PEEK and POKE allow direct access to any memory location. PEEK(i) returns the contents of the memory location specified by i. CALL POKE(i,b) causes the contents of the memory location specified by i to be replaced by the contents of b.

See also the IPEEK function in Appendix A.

RAN is another function in the FORTRAN library. RAN is a random number generator that is compatible with Microsoft's BASIC Compiler and BASIC-80 interpreter. The random number generated is a REAL decimal number between 0 and 1. The random number generator is called by a statement of the following form:

<variable> = RAN(x)

If x < 0, the first value of a new sequence of random numbers is returned.

If x = 0, the last random number generated is returned again.

If x > 0, the next random number in the sequence is generated.

Examples using library functions:

```
A1 = B+FLOAT (I7)

MAGNI = ABS(KBAR)

PDIF = DIM(C,D)

S3 = SIN(T12)

ROOT = (-B+SQRT(B**2-4.*A*C))/
1              (2.*A)
```

TABLE 9-1

Intrinsic Functions

Function Name	Definition	Types Argument	Types Function
ABS	|a|	Real	Real
IABS		Integer	Integer
DABS		Double	Double
AINT	Sign of a times lar-	Real	Real
INT	gest integer <=|a|	Real	Integer
IDINT		Double	Integer
AMOD	a1(mod a2)	Real	Real
MOD		Integer	Integer
AMAX0	Max(a1,a2,...)	Integer	Real
AMAX1		Real	Real
MAX0		Integer	Integer
MAX1		Real	Integer
DMAX1		Double	Double
AMIN0	Min(a1,a2,...)	Integer	Real
AMIN1		Real	Real
MIN0		Integer	Integer
MIN1		Real	Integer
DMIN1		Double	Double
FLOAT	Conversion from Integer to Real	Integer	Real
IFIX	Conversion from Real to Integer	Real	Integer
SIGN	Sign of a2 times |a1|	Real	Real
ISIGN		Integer	Integer
DSIGN		Double	Double
DIM	a1 - Min(a1,a2)	Real	Real
IDIM		Integer	Integer
SNGL		Double	Real
DBLE		Real	Double

TABLE 9-2

Basic External Functions

Name	Number of Arguments	Type Definition	Argument	Function
EXP	1	e**a	Real	Real
DEXP	1		Double	Double
ALOG	1	ln (a)	Real	Real
DLOG	1		Double	Double
ALOG10	1	log10(a)	Real	Real
DLOG10	1		Double	Double
SIN	1	sin (a)	Real	Real
DSIN	1		Double	Double
COS	1	cos (a)	Real	Real
DCOS	1		Double	Double
TANH	1	tanh (a)	Real	Real
SQRT	1	(a) ** 1/2	Real	Real
DSQRT	1		Double	Double
ATAN	1	arctan (a)	Real	Real
DATAN	1		Double	Double
ATAN2	2	arctan (a1/a2)	Real	Real
DATAN2	2		Double	Double
DMOD	2	a1(mod a2)	Double	Double

9.4 FUNCTION SUBPROGRAMS

A program unit which begins with a FUNCTION statement is called a FUNCTION subprogram.

A FUNCTION statement has one of the following forms:

```
t FUNCTION f(a1,a2,...an)
```

or

```
FUNCTION f(a1,a2,...an)
```

where:

1. t is either INTEGER, REAL, DOUBLE PRECISION or LOGICAL or is empty as shown in the second form.

2. f is the name of the FUNCTION subprogram.

3. The ai are dummy arguments of which there must be at least one and which represent variable names, array names or dummy names of SUBROUTINE or other FUNCTION subprograms.

9.5 CONSTRUCTION OF FUNCTION SUBPROGRAMS

Construction of FUNCTION subprograms must comply with the following restrictions:

1. The FUNCTION statement must be the first statement of the program unit.

2. Within the FUNCTION subprogram, the FUNCTION name must appear at least once on the left side of the equality sign of an assignment statement or as an item in the input list of an input statement. This defines the value of the FUNCTION so that it may be returned to the calling program.

   Additional values may be returned to the calling program through assignment of values to dummy arguments.

Example:

```
      FUNCTION Z7(A,B,C)
        .
        .
        .
      Z7 = 5.*(A-B) + SQRT(C)
        .
        .
        .
C     REDEFINE ARGUMENT
      B=B+Z7
        .
        .
        .
      RETURN
        .
        .
        .
      END
```

3. The names in the dummy argument list may not appear in EQUIVALENCE, COMMON or DATA statements in the FUNCTION subprogram.

4. If a dummy argument is an array name, then an array declarator must appear in the subprogram with dimensioning information consistant with that in the calling program.

5. A FUNCTION subprogram may contain any defined FORTRAN statements other than BLOCK DATA statements, SUBROUTINE statements, another FUNCTION statement or any statement which references either the FUNCTION being defined or another subprogram that references the FUNCTION being defined.

6. The logical termination of a FUNCTION subprogram is a RETURN statement and there must be at least one of them.

7. A FUNCTION subprogram must physically terminate with an END statement.

Example:

```
      FUNCTION SUM (BARY,I,J)
      DIMENSION BARY(10,20)
      SUM = 0.0
      DO 8 K=1,I
      DO8 M = 1,J
 8    SUM = SUM + BARY(K,M)
      RETURN
      END
```

9.6 REFERENCING A FUNCTION SUBPROGRAM

FUNCTION subprograms are called whenever the FUNCTION name, accompanied by an argument list, is used as an operand in an expression. Such references take the following form:

f(a1,a2,...,an)

where f is a FUNCTION name and the ai are actual arguments. Parentheses must be present in the form shown.

The arguments ai MUST agree in type, order and number with the dummy arguments in the FUNCTION statement of the called FUNCTION subprogram. There is no type conversion of arguments. There must be at least one argument. Arguments may be any of the following:

1. A variable name.

2. An array element name.

3. An array name.

4. An expression.

5. A SUBROUTINE or FUNCTION subprogram name.

6. A Hollerith or Literal constant.

If an ai is a subprogram name, that name must have previously been distinguished from ordinary variables by appearing in an EXTERNAL statement and the corresponding dummy arguments in the called FUNCTION subprograms must be used in subprogram references.

If ai is a Hollerith or Literal constant, the corresponding dummy variable should encompass enough storage units to correspond exactly to the amount of storage needed by the constant.

When a FUNCTION subprogram is called, program control goes to the first executable statement following the FUNCTION statement.

The following examples show references to FUNCTION subprograms.

```
Z10 = FT1+Z7(D,T3,RHO)

DIMENSION DAT(5,5)
.
.
.
S1 = TOT1 + SUM(DAT,5,5)
```

9.7 SUBROUTINE SUBPROGRAMS

A program unit which begins with a SUBROUTINE statement is called a SUBROUTINE subprogram. The SUBROUTINE statement has one of the following forms:

SUBROUTINE s (a1,a2,...,an)

or

SUBROUTINE s

where s is the name of the SUBROUTINE subprogram and each ai is a dummy argument which represents a variable or array name or another SUBROUTINE or FUNCTION name.

9.8 CONSTRUCTION OF SUBROUTINE SUBPROGRAMS

1. The SUBROUTINE statement must be the first statement of the subprogram.

2. The SUBROUTINE subprogram name must not appear in any statement other than the initial SUBROUTINE statement.

3. The dummy argument names must not appear in EQUIVALENCE, COMMON or DATA statements in the subprogram.

4. If a dummy argument is an array name then an array declarator must appear in the subprogram with dimensioning information consistant with that in the calling program.

5. If any of the dummy arguments represent values that are to be determined by the SUBROUTINE subprogram and returned to the calling program, these dummy arguments must appear within the subprogram on the left side of the equality sign in a replacement statement, in the input

list of an input statement or as a parameter within a subprogram reference.

6. A SUBROUTINE may contain any FORTRAN statements other than BLOCK DATA statements, FUNCTION statements, another SUBROUTINE statement, a PROGRAM statement or any statement which references the SUBROUTINE subprogram being defined or another subprogram which references the SUBROUTINE subprogram being defined.

7. A SUBROUTINE subprogram may contain any number of RETURN statements. It must have at least one.

8. The RETURN statement(s) is the logical termination point of the subprogram.

9. The physical termination of a SUBROUTINE subprogram is an END statement.

10. If an actual argument transmitted to a SUBROUTINE subprogram by the calling program is the name of a SUBROUTINE or FUNCTION subprogram, the corresponding dummy argument must be used in the called SUBROUTINE subprogram as a subprogram reference.

Example:

```
C  SUBROUTINE TO COUNT POSITIVE ELEMENTS
C          IN AN ARRAY
       SUBROUTINE COUNT P(ARRY,I,CNT)
       DIMENSION ARRY(7)
       CNT = 0
       DO 9 J=1,I
       IF(ARRY(J))9,5,5
   9   CONTINUE
       RETURN
   5   CNT = CNT+1.0
       GO TO 9
       END
```

9.9 REFERENCING A SUBROUTINE SUBPROGRAM

A SUBROUTINE subprogram may be called by using a CALL statement. A CALL statement has one of the following forms:

```
CALL s(a1,a2,...,an)
```

or

```
CALL s
```

where s is a SUBROUTINE subprogram name and the ai are the actual arguments to be used by the subprogram. The ai must agree in type, order and number with the corresponding dummy arguments in the subprogram-defining SUBROUTINE statement.

The arguments in a CALL statement must comply with the following rules:

1. FUNCTION and SUBROUTINE names appearing in the argument list must have previously appeared in an EXTERNAL statement.

2. If the called SUBROUTINE subprogram contains a variable array declarator, then the CALL statement must contain the actual name of the array and the actual dimension specifications as arguments.

3. If an item in the SUBROUTINE subprogram dummy argument list is an array, the corresponding item in the CALL statement argument list must be an array.

When a SUBROUTINE subprogram is called, program control goes to the first executable statement following the SUBROUTINE statement.

Example:

```
      DIMENSION DATA(10)
      .
      .
      .
C  THE STATEMENT BELOW CALLS THE
C       SUBROUTINE IN THE PREVIOUS PARAGRAPH
C
      CALL COUNTP(DATA,10,CPOS)
```

9.10 RETURN FROM FUNCTION AND SUBROUTINE SUBPROGRAMS

The logical termination of a FUNCTION or SUBROUTINE subprogram is a RETURN statement which transfers control back to the calling program. The general form of the RETURN statement is simply the word

RETURN

The following rules govern the use of the RETURN statement:

1. There must be at least one RETURN statement in each SUBROUTINE or FUNCTION subprogram.

2. RETURN from a FUNCTION subprogram is to the instruction sequence of the calling program following the FUNCTION reference.

3. RETURN from a SUBROUTINE subprogram is to the next executable statement in the calling program which would logically follow the CALL statement.

4. Upon return from a FUNCTION subprogram the single-valued result of the subprogram is available to the evaluation of the expression from which the FUNCTION call was made.

5. Upon return from a SUBROUTINE subprogram the values assigned to the arguments in the SUBROUTINE are available for use by the calling program.

Example:

```
Calling Program Unit
.
.
.
    CALL SUBR(Z9,B7,R1)
.
.
.

Called Program Unit

    SUBROUTINE SUBR(A,B,C)
    READ(3,7) B
    A = B**C
    RETURN
7   FORMAT(F9.2)
    END
```

In this example, Z9 and B7 are made available to the calling program when the RETURN occurs.

9.11 PROCESSING ARRAYS IN SUBPROGRAMS

If a calling program passes an array name to a subprogram, the subprogram must contain the dimension information pertinent to the array. A subprogram must contain array declarators if any of its dummy arguments represent arrays or array elements.

For example, a FUNCTION subprogram designed to compute the average of the elements of any one dimension array might be the folowing:

```
        Calling Program Unit

        DIMENSION Z1(50),Z2(25)
        .
        .
        .
        A1 = AVG(Z1,50)
        .
        .
        .
        A2 = A1-AVG(Z2,25)
        .
        .
        .

        Called Program Unit

        FUNCTION AVG(ARG,I)
        DIMENSION ARG(50)
        SUM = 0.0
        DO 20 J=1,I
   20   SUM = SUM + ARG(J)
        AVG = SUM/FLOAT(I)
        RETURN
        END
```

Note that actual arrays to be processed by the FUNCTION subprogram are dimensioned in the calling program and the array names and their actual dimensions are transmitted to the FUNCTION subprogram by the FUNCTION subprogram reference. The FUNCTION subprogram itself contains a dummy array and specifies an array declarator.

Dimensioning information may also be passed to the subprogram in the paramater list. For example:

```
Calling Program Unit

DIMENSION A(3,4,5)
.
.
.
CALL SUBR(A,3,4,5)
.
.
.
END

Called Program Unit

SUBROUTINE SUBR(X,I,J,K)
DIMENSION X(I,J,K)
.
.
.
RETURN
END
```

It is valid to use variable dimensions only when the array name and all of the variable dimensions are dummy arguments. The variable dimensions must be type Integer. It is invalid to change the values of any of the variable dimensions within the called program.

9.12 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram has as its only purpose the initialization of data in a COMMON block during loading of a FORTRAN object program. BLOCK DATA subprograms begin with a BLOCK DATA statement of the following form:

BLOCK DATA [subprogram-name]

and end with an END statement. Such subprograms may contain only Type, EQUIVALENCE, DATA, COMMON and DIMENSION statements and are subject to the following considerations:

1. If any element in a COMMON block is to be initialized, all elements of the block must be listed in the COMMON statement even though they might not all be initialized.

2. Initialization of data in more than one COMMON block may be accomplished in one BLOCK DATA subprogram.

3. There may be more than one BLOCK DATA subprogram loaded at any given time.

4. Any particular COMMON block item should only be initialized by one program unit.

Example:

```
      BLOCK DATA
      LOGICAL A1
      COMMON/BETA/B(3,3)/GAM/C(4)
      COMMON/ALPHA/A1,F,E,D
      DATA B/1.1,2.5,3.8,3*4.96,
     12*0.52,1.1/,C/1.2E0,3*4.0/
      DATA A1/.TRUE./,E/-5.6/
```

9.13 PROGRAM CHAINING

Programs may be loaded and executed (CHAINed) by a FORTRAN program through the CALL FCHAIN facility. Variables are not preserved from one program to the next. The syntax is:

```
CALL FCHAIN ('filename',drive)
```

where

Filename = an ASCII name which the operating system will associate with the file. The Filename should be a Hollerith or Literal constant, or a variable or array name, where the variable or array contains the ASCII name. The Filename should be blank-filled to exactly eleven characters.

Drive = the disk drive on which the file exists or will exist (must be an Integer constant or Integer variable within the range allowed by the operating system).

RULES:

1. The program CHAINed must be a "MAIN" program. That is, one having an ENTRY Point. FORTRAN, COBOL, and assembly language subroutines do not conntain a "MAIN" entry point.

2. Parameters may not be passed to CHAINed programs.

3. Illegal filename, Illegal drive specification, File not found, Out of memory, and Disk read errors will result in a fatal **IO** Error.

Examples:

```
CALL FCHAIN('PROG2   COM',0)
```

Loads PROG2.COM from the currently logged disk and executes it.

```
CALL FCHAIN('PROG2   OLD',2)
```

Loads PROG2.OLD from drive B: (PROG2.OLD must be a COM file) and executes it.

Note that "0" indicates the currently logged drive, "1" indicates drive A:, and so on. Note that the filename and extension use exactly 11 spaces. The filename is left-justified in the first 8 spaces, the extension is right-justified in the last 3

spaces, and blanks are inserted to make 11 characters.

APPENDIX A

FORTRAN-80 Language Extensions

A.1 New FORTRAN-80 Features, Apple II Version

FORTRAN-80 for the Apple II Computer has many unique features that take advantage of the Apple II hardware and CP/M operating system. These special features are described in this appendix.

All of the special Apple II features are implemented as subroutines or functions.

NOTE: Except where otherwise noted, only BYTE and INTEGER values can be used as parameters to these subroutines and functions. If INTEGER values are used, only the low byte is significant.

NOTE: No range check is performed on any of the subroutines and functions. Therefore, assure that the values supplied are within the proper ranges to avoid unpredictable results.

Low Resolution Graphics

Only low resolution graphics features (40x48 screen) are provided. High resolution graphics capabilities are not available with FORTRAN-80.

1. GR Subroutine

Syntax: CALL GR(M,C)

The GR subroutine initializes the screen for low resolution graphics.

M is the screen mode.

M=0 initializes a 40x40 lores graphics mode screen. The bottom 4 lines of the screen are left in text mode.

M=1 initializes the entire screen, 40x48, for lores graphics.

The initialized screen is filled with color C. C must be in the range 0 to 15 (see COLOR Subroutine below).

2. COLOR Subroutine

Syntax: CALL COLOR(C)

The COLOR subroutine sets the color for all subsequent lores plotting. C must be in the range 0 to 15, with colors numbered as follows:

0 black	6 medium blue	11 pink
1 magenta	7 light blue	12 green
2 dark blue	8 brown	13 yellow
3 purple	9 orange	14 aqua
4 dark green	10 grey	15 white
5 grey		

3. PLOT Subroutine

Syntax: CALL PLOT(X,Y)

The PLOT subroutine plots a point at location X,Y on the lores screen. X must be in the range 0 to 39. Y must be in the range 0 to 47. The point (0,0) is at the top left corner of the screen, and (39,47) is at the lower right corner. The point is plotted in the color set by the most recent COLOR or GR call.

4. HLIN Subroutine

Syntax: CALL HLIN(X0,X1,Y)

The HLIN subroutine draws a horizontal line from X0,Y to X1,Y. X0 must be less than X1.

5. VLIN Subroutine

Syntax: CALL VLIN(Y0,Y1,X)

The VLIN subroutine draws a vertical line from X,Y0 to X,Y1. Y0 must be less than Y1.

6. SCRN Function

Syntax: C=SCRN(X,Y)

The SCRN function returns the screen color at point X,Y.

Paddles and Beeps

1. PDL Function

Syntax: P=PDL(N)
where N is he paddle number (0 to 3)

The PDL function returns the current paddle value (0 to 255) of paddle N.

2. BUTTON Subroutine

Syntax: B=BUTTON(N)
where N is the button number (0 to 3)

The BUTTON function returns the status of button N. It returns -1 (.TRUE.) if the button is pressed, and 0 (.FALSE.) if it is not pressed.

3. BEEP Subroutine

Syntax: CALL BEEP(P,D)

The BEEP subroutine causes a tone to be emitted. The tone has pitch P and duration D. The highest pitch is 0 and the lowest is 255. The shortest duration is 0 and the longest is 255. BEEP is intended for sound-effect purposes only.

Direct Cursor Positioning

1. GOTOXY Subroutine

   Syntax: CALL GOTOXY(X,Y)

   The GOTOXY subroutine positions the cursor at point X,Y on the screen. To begin WRITEing at that point, the next formatted WRITE statement must have the "*" carriage control as the first character. (See Section 8.7.9 of the FORTRAN-80 Reference Manual.)

2. HOME Subroutine

   Syntax: CALL HOME

   The HOME subroutine clears the screen and positions the cursor at the top left corner of the screen.

3. SCREEN Subroutine

   Syntax: CALL SCREEN(F)

   Executes the IOCB screen function denoted by F. F is in the range 1 to 9:

   1 Clear screen
   2 Clear to end of page
   3 Clear to end of line
   4 Set normal (low-light) text mode
   5 Set inverse (high-light) text mode
   6 Return cursor to upper left corner
   7 Address cursor
   8 Move cursor up one line
   9 Non-destructively move cursor forward

   CALL SCREEN(1) is equivalent to CALL HOME.

Single-Character I/O

1. INKEY Function

   Syntax: C=INKEY(M)

   If M=0, INKEY returns 0 (.FALSE.) if no character is available at the keyboard; or it returns the ASCII value of the character if a character has been typed.

   If M=1, INKEY waits until the next character is typed at the keyboard, and then returns the ASCII value of that character.

2. CONOUT Subroutine

   Syntax: CALL CONOUT(C)

   The CONOUT subroutine outputs the character C to the console.

3. PUNCH Subroutine

   Syntax: CALL PUNCH(C)

   The PUNCH subroutine outputs byte C to the CP/M PUN: device (Apple slot 2). C must be a variable of type BYTE.

4. READER Function

   Syntax: C=READER(0)

   The READER function returns a single character from the CP/M RDR: device (Apple slot 2). Zero is a dummy argument, and has no meaning. An argument must be included so that the compiler will recognize READER as a function.

Miscellaneous Functions

1. UNSIGN Function

   Syntax: I=UNSIGN(B)

   The UNSIGN function converts byte value B to an unsigned integer value. Thus, if B is -1, UNSIGN returns 255. If B is -128, UNSIGN returns 128, and so on.

2. IPEEK Function

   Syntax: I=IPEEK(A)

   The IPEEK function returns the contents of address A as an unsigned integer. Equivalent to I=UNSIGN(PEEK(A))

A.2 FORTRAN-80 Extensions to the ANSI Standard

The FORTRAN-80 language includes the following extensions to ANSI Standard FORTRAN (X3.9-1966).

1. If c is used in a 'STOP c' or 'PAUSE c' statement, c may be any six ASCII characters.

2. Error and End-of-File branches may be specified in READ and WRITE statements using the ERR= and END= options.

3. The standard subprograms PEEK and POKE have been added to the FORTRAN library.

4. Statement functions may use subscripted variables.

5. Hexadecimal constants may be used wherever Integer constants are normally allowed.

6. The literal form of Hollerith data (character string between apostrophe characters) is permitted in place of the standard nH form.

7. Holleriths and Literals are allowed in expressions in place of Integer constants.

8. There is no restriction to the number of continuation lines.

9. Mixed mode expressions and assignments are allowed, and conversions are done automatically.

A.3 FORTRAN-80 Language Restrictions

FORTRAN-80 places the following restrictions upon Standard FORTRAN.

1. The COMPLEX data type is not implemented.
2. The specification statements must appear in the following order:
    1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
    2. Type, EXTERNAL, DIMENSION
    3. COMMON
    4. EQUIVALENCE
    5. DATA
    6. Statement Functions
3. A different amount of computer memory is allocated for each of the data types: Integer, Real, Double Precision, Logical.
4. The equal sign of a replacement statement and the first comma of a DO statement must appear on the initial statement line.
5. In Input/Output list specifications, sublists enclosed in parentheses are not allowed.

Descriptions of these language extensions and restrictions are included at the appropriate points in the text of this document.

APPENDIX B

I/O Interface

NOTE: Modification and assembly of $LUNTB or any assembly language library routines requires the use of Microsoft's MACRO-80 Assembler and LIB-80 Library Manager.

Input/Output operations are table-dispatched to the driver routine for the proper Logical Unit Number. $LUNTB is the dispatch table. It contains one 2-byte driver address for each possible LUN. It also has a one-byte entry at the beginning, which contains the maximum LUN plus one. The initial run-time package provides for 10 LUNs (1-10). Any of these may be redefined by the user simply by changing the appropriate entries in $LUNTB. The runtime system uses LUN 3 for errors and other user communication. Therefore, LUN 3 should correspond to the operator console.

The device drivers also contain local dispatch tables. Note that $LUNTB contains one address for each device, yet there are really seven possible operations per device:

1) Formatted Read
2) Formatted Write
3) Unformatted Read
4) Unformatted Write
5) Rewind
6) Backspace
7) Endfile

Each device driver contains up to seven routines. The starting addresses of each of these seven routines are placed at the beginning of the driver, in the exact order listed above. The entry in $LUNTB then points to this local table, and the runtime system indexes into it to get the address of the appropriate routine to handle the requested I/O operation.

The following conventions apply to the individual I/O routines:

1. Location $BF contains the data buffer address for READs and WRITEs.

2. For a WRITE, the number of bytes to write is in location $BL.

3. For a READ, the number of bytes read should be returned in $BL.

4. All I/O operations set the condition codes before exit to indicate an error condition, end-of-file condition, or normal return:

   a) CY=1, Z=don't care - I/O error
   b) CY=0, Z=0 - end-of-file encountered
   c) CY=0, Z=1 - normal return

   The runtime system checks the condition codes after calling the driver. If they indicate a non-normal condition, control is passed to the label specified by "ERR=" or "END=" or, if no label is specified, a fatal error results.

5. $IOERR is a global routine which prints an "ILLEGAL I/O OPERATION" message (non-fatal). This routine may be used if there are some operations not allowed on a particular device (i.e. Unformatted I/O on a TTY).

NOTE

The I/O buffer has a fixed maximum length of 132 bytes. If a driver allows an input operation to write past the end of the buffer, essential runtime variables may be affected. The consequences are unpredictable.

APPENDIX C

Subprogram Linkages

NOTE: This discussion of assembly language subroutines assumes that the user has access to an assembler, such as Microsoft's MACRO-80 assembler. MACRO-80 is not supplied with the FORTRAN-80 package. MACRO-80 is available from Microsoft Consumer Products.

This appendix defines a normal subprogram call as generated by the FORTRAN compiler. It is included to facilitate linkages between FORTRAN programs and those written in other languages, such as 8080 Assembly.

A subprogram reference with no parameters generates a simple "CALL" instruction. The corresponding subprogram should return via a simple "RET." (CALL and RET are 8080 opcodes - see the assembly manual or 8080 reference manual for explanations.)

A subprogram reference with parameters results in a somewhat more complex calling sequence. Parameters are always passed by reference (i.e., the thing passed is actually the address of the low byte of the actual argument). Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter 1 in HL.

    2. Parameter 2 in DE.

    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subprogram must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. Neither the compiler nor

the runtime system checks for the correct number of parameters.

If the subprogram expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT, and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subprogram is responsible for saving the first two parameters before calling $AT. For example, if a subprogram expects 5 parameters, it should look like:

```
SUBR:   SHLD    P1          ;SAVE PARAMETER 1
        XCHG
        SHLD    P2          ;SAVE PARAMETER 2
        MVI     A,3         ;NO. OF PARAMETERS LEFT
        LXI     H,P3        ;POINTER TO LOCAL AREA
        CALL    $AT         ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subprogram]
        .
        .
        .
        RET                 ;RETURN TO CALLER
P1:     DS      2           ;SPACE FOR PARAMETER 1
P2:     DS      2           ;SPACE FOR PARAMETER 2
P3:     DS      6           ;SPACE FOR PARAMETERS 3-5
```

When accessing parameters in a subprogram, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subprogram. This applies to FORTRAN subprograms, as well as those written in assembly language.

FORTRAN Functions (Section 9) return their values in registers or memory depending upon the type. Logical results are returned in (A), Integers in (HL). Extended integers and Reals return results in memory at $AC. Double Precision Reals return results in memory at $DAC. $AC and

$DAC are the addresses of the low bytes of the mantissas.

APPENDIX D

ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

APPENDIX E

Referencing FORTRAN-80 Library Subroutines

NOTE: This discussion of assembly language subroutines assumes that the user has access to an assembler, such as Microsoft's MACRO-80 assembler. MACRO-80 is not supplied with the FORTRAN-80 package. MACRO-80 is available from Microsoft Consumer Products.

The FORTRAN-80 library contains a number of subroutines that may be referenced by the user from FORTRAN or assembly programs.

1. Referencing Arithmetic Routines

In the following descriptions, $AC refers to the floating accumulator; $AC is the address of the low byte of the mantissa. $AC+3 is the address of the exponent. $DAC refers to the DOUBLE PRECISION accumulator; $DAC is the address of the low byte of the mantissa. $DAC+7 is the address of the DOUBLE PRECISION exponent.

All arithmetic routines (addition, subtraction, multiplication, division, exponentiation) adhere to the following calling conventions.

1. Argument 1 is passed in the registers:
   Integer in [HL]
   Real in $AC
   Double in $DAC

2. Argument 2 is passed either in registers, or in memory depending upon the type:

   a. Integers are passed in [HL], or [DE] if [HL] contains Argument 1.

   b. Real and Double Precision values are passed in memory pointed to by [HL]. ([HL] points to the low byte of the mantissa.)

The following arithmetic routines are contained in the Library:

Function	Name	Argument 1 Type	Argument 2 Type
Addition	$AY	Integer*4	Integer
	$A1	Integer*4	Integer*4
	$AA	Real	Integer
	$AE	Real	Integer*4
	$AB	Real	Real
	$AQ	Double	Integer
	$AV	Double	Integer*4
	$AR	Double	Real
	$AU	Double	Double
Division	$D9	Integer	Integer
	$DY	Integer*4	Integer
	$D1	Integer*4	Integer*4
	$DA	Real	Integer
	$DE	Real	Integer*4
	$DB	Real	Real
	$DQ	Double	Integer
	$DV	Double	Integer*4
	$DR	Double	Real
	$DU	Double	Double
Exponentiation	$E9	Integer	Integer
	$EY	Integer*4	Integer
	$E1	Integer*4	Integer*4
	$EA	Real	Integer
	$EE	Real	Integer*4
	$EB	Real	Real
	$EQ	Double	Integer
	$EV	Double	Integer*4
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MY	Integer*4	Integer
	$M1	Integer*4	Integer*4
	$MA	Real	Integer
	$ME	Real	Integer*4
	$MB	Real	Real
	$MQ	Double	Integer
	$MV	Double	Integer*4
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SY	Integer*4	Integer
	$S1	Integer*4	Integer*4
	$SA	Real	Integer
	$SE	Real	Integer*4
	$SB	Real	Real
	$SQ	Double	Integer
	$SV	Double	Integer*4
	$SR	Double	Real
	$SU	Double	Double

Additional Library routines are provided for converting between value types. Arguments are always passed to and returned by these conversion routines in the appropriate registers:

Logical in [A]

Integer in [HL]

Extended Integer in $AC

Real in $AC

Double Precision REAL in $DAC

Name	Function
$CD	Integer to Integer*4
$CA	Integer to Real
$CC	Integer to Double
$C4	Integer*4 to Integer
$C5	Integer*4 to Real
$C6	Integer*4 to Logical
$C7	Integer*4 to Double
$CH	Real to Integer
$CL	Real to Integer*4
$CJ	Real to Logical
$CK	Real to Double
$CX	Double to Integer
$C0	Double to Integer*4
$CY	Double to Real
$CZ	Double to Logical

2. Referencing Intrinsic Functions

Intrinsic Functions are passed their parameters in H,L and D,E. If there are three arguments, B,C contains the third parameter. If there are more than three arguments, B,C contains a pointer to a block in memory that holds the remaining parameters. Each of these parameters is a pointer to an argument. (See Appendix B.)

For a MIN or MAX function, the number of arguments is passed in A.

NOTE

None of the functions (except INP and OUT) may take a byte variable as an argument. Byte variables must first be converted to the type expected by the function. Otherwise, results will be unpredictable.

3. Formatted READ and WRITE Routines

A READ or WRITE statement calls one of the following routines:

$W2 (2 parameters) $W5 (5 parameters)	Initialize for an I/O transfer to a device (WRITE)
$R2 (2 parameters) $R5 (5 parameters)	Initialize for an I/O transfer from a device (READ)

These routines adhere to the following calling conventions:

1. H,L points to the LUN
2. D,E points to the beginning of the FORMAT statement
3. If the routine has five parameters, then B,C points to a block of three parameters:
    a. the address for an ERR= branch
    b. the address for an EOF= branch
    c. the address for a REC= value

The routines that transfer values into the I/O buffer are:

$I0	transfers integers
$I1	transfers real numbers
$I2	transfers logicals
$I3	transfers double precision numbers
$I4	transfers extended integers (4 bytes)

Transfer routines adhere to the following calling conventions:

1. H,L points to a location that contains the number of dimensions for the variables in the list

2. D,E points to the first value to be transferred

3. B,C points to the second value to be transferred if there are exactly two values to be transferred by this call. If there are more than two values, B,C points to a block that contains pointers to the second through nth values.

4. Register A contains the number of parameters (including H,L) generated by this call.

The routine $ND terminates the I/O process.

Example:

```
        EXTRN   $W2,$IO,$ND
        ENTRY   TEST
TEST:   LXI     H,LUN
        LXI     D,FORMAT
        CALL    $W2

        LXI     H,DIMENS
        LXI     D,NUMBER
        MVI     A,2
        CALL    $IO

        CALL    $ND

        RET
LUN:    DW      1
FORMAT: DB      '(11H RESULT IS=,15)'
DIMENS: DW      1
NUMBER: DW      9999

        END     TEST
```

4. Loading and Storing Floating Accumulator

In the following definitions, $AC refers to the floating accumulator and $DAC refers to the DOUBLE PRECISION accumulator.

To Load Floating Accumulator:
(H,L points to value to be loaded.)

Name	Function
$L1	Loads into $AC, 4 bytes
$L3	Loads into $DAC, 8 bytes

To Store Floating Accumulator:
(H,L points to memory where value is to be stored)

Name	Function
$T1	Stores 4 bytes from $AC
$T3	Stores 8 bytes from $DAC

INDEX

Apple II 105
Arithmetic Expression . . 24-25, 48
Arithmetic IF 43, 48, 50
Array 13, 19, 33-34, 36-37, 39-40, 59, 84, 93, 95, 100-101
Array Declarator 19
Array Element 13, 19, 26, 31, 38
ASCII Character Codes . . 118
ASSIGN 43, 46-47
Assigned GOTO 43, 46

BACKSPACE 65
BEEP subroutine 107
BLOCK DATA 33, 36, 97, 102
BUTTON subroutine 107

CALL 43, 55, 98
Characteristic 22
COLOR subroutine 106
Comment Line 9
COMMON 33, 37-40, 93, 96, 102
Computed GOTO 43, 45
CONOUT subroutine 109
Constant 13-14
Continuation 8-9
CONTINUE 43, 53
Control Statements . . . 43
Cursor positioning . . . 108

DATA 33, 40, 93, 96, 102
Data Representation . . . 13
Data Storage 21
DECODE 65
DIMENSION 19, 33, 36, 102
Disk Files 62
DO 43, 49-51
DO Implied List 67
Double precision 13, 23
Dummy 96-98, 100-101

ENCODE 65
END 43, 55, 93, 97, 102
END Line 10
ENDFILE 65
EQUIVALENCE 33, 38-40, 93, 96, 102
Executable 11, 33, 43
Expression 24-25, 30-31
Extended Integer 14
Extended Range 52

EXTERNAL 33, 36, 95, 98
External Functions . . . 91

Field Descriptors 70
FORMAT 58-60, 69, 73, 75-85
Formatted READ 57
Formatted WRITE 60
FUNCTION 33, 36, 86, 92-93, 95-99, 101

GOTO 43-44, 50
GOTOXY subroutine 108
GR subroutine 106
Graphics 106

Hexadecimal 21, 30, 41
HLIN subroutine 107
Hollerith 14, 19, 21, 30, 41, 59, 75-77, 95
HOME subroutine 108

I/O 56, 113
I/O List 66
IF 43, 48
IMPLICIT 42
INCLUDE 12
Index 50
Initial Line 8
INKEY function 109
Integer 13, 18, 22
Integer*2 21
Integer*4 . . . . . 14, 17, 21, 23
Intrinsic Functions . . . 90, 121
IPEEK function 110

Label 8, 10, 43-44, 50
Library Function 86, 88
Library Subroutines . . . 119
Line Format 8
List Item 66
Literal 19, 21, 30, 41, 76-77, 95
Logical 13, 18, 22, 77
Logical Expression . . . 26, 29, 49
Logical IF 43, 49-50
Logical Operator 28
Logical Unit Number . . . 57, 61, 113
LUN 57, 61, 113

Mantissa 22

Nested 52
Non-executable 11, 33
Numeric Conversions . . . 71

OPEN 64
Operand 24
Operator 24

Paddles 107
PAUSE 43, 50, 54
PDL function 107
PEEK 89
PLOT subroutine 106
POKE 89
PROGRAM 33, 87, 97
PUNCH subroutine 109

RAN 89
Range 50
READ 59, 61, 69, 79, 82-85, 122
READER function 109
Real 13, 18, 22
Relational Expression . . 26-27
Relational Operator . . . 27
Replacement Statement . . 31, 49
RETURN 43, 50, 55, 93, 97, 99
REWIND 65

Scale Factor 79
SCREEN subroutine 108
SCRN function 107
Specification Statement . 33
Statement Function . . . 33, 86-87
STOP 43, 50, 54
Storage 34
Storage Format 13
Storage Unit 21-22, 38
Subprogram 36, 55, 86, 92-93, 95-102, 115
SUBROUTINE 33, 36, 55, 86, 93, 95-99
Subscript 19-20, 26
Subscript Expression . . 20, 26

Type 102
Type Statement 34

Unconditional GOTO . . . 44
Unformatted I/O 61
UNSIGN function 110

Variable 13, 18, 31, 37, 95
VLIN subroutine 107

WRITE 60-61, 69, 79, 82-85, 122

Microsoft
Software Problem Report

Use this form to report errors or problems in: ☐ FORTRAN-80
☐ COBOL-80
☐ MACRO-80
☐ LINK-80

Release (or version) number: _____

Date _____

Report only one problem per form.

Describe your hardware and operating system: _____

_____

Please supply a concise description of the problem and the circumstances surrounding its occurrence. If possible, reduce the problem to a simple test case. Otherwise, include all programs and data in machine readable form (preferably on a diskette). If a patch or interim solution is being used, please describe it.

This form may also be used to describe suggested enhancements to Microsoft software.

Problem Description:

Did you find errors in the documentation supplied with the software? If so, please include page numbers and describe:

Fill in the following information before returning this form:

Name_____ Phone_____

Organization_____

Address_____ City_____ State____ Zip_____

Return form to:

Microsoft
10800 NE Eighth, Suite 819
Bellevue, WA 98004

[illegible]

FORTRAN-80

MICROSOFT